Titre: Title:	Parallelization Strategies for Modern Computing Platforms: Application to Illustrative Image Processing and Computer Vision Applications
Auteur: Author:	Taieb Lamine Ben Cheikh
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ben Cheikh, T. L. (2015). Parallelization Strategies for Modern Computing Platforms: Application to Illustrative Image Processing and Computer Vision Applications [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1733/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1733/
Directeurs de recherche: Advisors:	Gabriela Nicolescu, & Sofiene Tahar
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

PARALLELIZATION STRATEGIES FOR MODERN COMPUTING PLATFORMS: APPLICATION TO ILLUSTRATIVE IMAGE PROCESSING AND COMPUTER VISION APPLICATIONS

TAIEB LAMINE BEN CHEIKH
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

PARALLELIZATION STRATEGIES FOR MODERN COMPUTING PLATFORMS: APPLICATION TO ILLUSTRATIVE IMAGE PROCESSING AND COMPUTER VISION APPLICATIONS

présentée par : BEN CHEIKH Taieb Lamine
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. DAGENAIS Michel, Ph. D, président
Mme NICOLESCU Gabriela, Doctorat, membre et directrice de recherche
M. TAHAR Sofiene, Ph. D., membre et codirecteur de recherche
Mme CHERIET Farida, Ph. D., membre
M. ZENG Haibo, Ph. D., membre externe

DEDICATION

I dedicate my Ph.D work to

My parents : ***Beya*** *and* ***Khemaies****,*

My family : ***Leila****,* ***Mohamed Ali****,* ***Walid*** *and* ***Olfa***

My darling nieces and nephews : ***Skander****,* ***Norhene****,* ***Omar****,* ***Houssem****,* ***Nour*** *and* ***Yesmine***

My family in law : ***Habib****,* ***Feten****,* ***Sonda****,* ***Omar*** *and* ***Fekher***

My professors in Tunisia from primary school to the University

And last but not least,

To you, my ***Hana****, my wife :*
We did it !
You always make me feel on top of things...

———

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor **Dr. Gabriela Nicolescu** for all of the guidance she has given me throughout my Ph.D studies. I am indebted to her for all of the time and patience she extended to me to complete my degree. I acknowledge you for your lasting support through this whole period.
I would also like to thank my thesis co-director **Dr. Sofiène Tahar** for his interesting discussions around my work.
An appreciative thanks is extended to the members of the jury who accepted to read, examine and evaluate my work.

I would like to express thanks to **Dr. Giovani Beltrame**, **Dr. Jelena Trajkovic**, for all of their assistance and the critical reading of my academic reports during my Ph.D. Thanks for the many fun hours we have spent discussing my work.
To **ST Microelectronics** team and **CMC Microsystems** team, especially **Pierre Paulin**, **Youcef Bouchebaba** and **Yassine Hariri**. I greatly appreciate your collaboration in the project. Thank you for your critical discussions of my results, it is always beneficial to get another perspective on things.

A special thought goes to **Rabeh**, **Alain**, and **Felippe**, with whom I shared the office, for great support and friendship during these years. A special appreciation goes to **Mahdi** and **Alexandra** ; who have challenged and inspired me through their critical reading of my thesis.

To you my best friend **Imen**, thank you for the motivation and close friendship we have developed over these years- You have been and you're still a huge support. To you guys : **Mohamed**, **Rafik**, **Jihed**, **Wael**, **Sami**, **Haithem** and **Rachid**. I want to thank you not only for sharing hours of Tennis-Volleyball-Football or Ping-Pong :) but also for your close and good friendship. I am happy to have all of you and to be able to share both the hard but also the enjoyable times with.

I am greatly thankful to the staff of Ecole Polytechnique de Montreal ; all contributed to the environment and atmosphere of the University, making it an exciting place to come to every day, be it for work or socializing, and often both.

RÉSUMÉ

L'évolution spectaculaire des technologies dans le domaine du matériel et du logiciel a permis l'émergence des nouvelles plateformes parallèles très performantes. Ces plateformes ont marqué le début d'une nouvelle ère de la computation et il est préconisé qu'elles vont rester dans le domaine pour une bonne période de temps. Elles sont présentes déjà dans le domaine du calcul de haute performance (en anglais HPC, High Performance Computer) ainsi que dans le domaine des systèmes embarqués. Récemment, dans ces domaines le concept de calcul hétérogène a été adopté pour atteindre des performances élevées. Ainsi, plusieurs types de processeurs sont utilisés, dont les plus populaires sont les unités centrales de traitement ou CPU (de l'anglais Central Processing Unit) et les processeurs graphiques ou GPU (de l'anglais Graphics Processing Units).

La programmation efficace pour ces nouvelles plateformes parallèles amène actuellement non seulement des opportunités mais aussi des défis importants pour les concepteurs. Par conséquent, l'industrie a besoin de l'appui de la communauté de recherche pour assurer le succès de ce nouveau changement de paradigme vers le calcul parallèle. Trois défis principaux présents pour les processeurs GPU massivement parallèles (ou "many-cores") ainsi que pour les processeurs CPU multi-coeurs sont : (1) la sélection de la meilleure plateforme parallèle pour une application donnée, (2) la sélection de la meilleure stratégie de parallèlisation et (3) le réglage minutieux des performances (ou en anglais performance tuning) pour mieux exploiter les plateformes existantes.

Dans ce contexte, l'objectif global de notre projet de recherche est de définir de nouvelles solutions pour aider à la programmation efficace des applications complexes sur les plateformes parallèles modernes.

Les principales contributions à la recherche sont :

1. L'évaluation de l'efficacité d'accélération pour plusieurs plateformes parallèles, dans le cas des applications de calcul intensif.
2. Une analyse quantitative des stratégies de parallélisation et implantation sur les plateformes à base de processeurs CPU multi-coeur ainsi que pour les plateformes à base de processeurs GPU massivement parallèles.
3. La définition et la mise en place d'une approche de réglage de performances (en Anglais performance tuning) pour les plateformes parallèles.

Les contributions proposées ont été validées en utilisant des applications réelles illustratives et un ensemble varié de plateformes parallèles modernes.

ABSTRACT

With the technology improvement for both hardware and software, parallel platforms started a new computing era and they are here to stay. Parallel platforms may be found in High Performance Computers (HPC) or embedded computers. Recently, both HPC and embedded computers are moving toward heterogeneous computing platforms. They are employing both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to achieve the highest performance. Programming efficiently for parallel platforms brings new opportunities but also several challenges. Therefore, industry needs help from the research community to succeed in its recent dramatic shift to parallel computing.

Parallel programing presents several major challenges. These challenges are equally present whether one programs on a many-core GPU or on a multi-core CPU. Three of the main challenges are: (1) Finding the best platform providing the required acceleration (2) Select the best parallelization strategy (3) Performance tuning to efficiently leverage the parallel platforms.

In this context, the overall objective of our research is to propose a new solution helping designers to efficiently program complex applications on modern parallel architectures. The contributions of this thesis are:

1. The evaluation of the efficiency of several target parallel platforms to speedup compute intensive applications.
2. The quantitative analysis for parallelization and implementation strategies on multi-core CPUs and many-core GPUs.
3. The definition and implementation of a new performance tuning framework for heterogeneous parallel platforms.

The contributions were validated using real computation intensive applications and modern parallel platform based on multi-core CPU and many-core GPU.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

APU	Accelerated Processing Unit
ALU	Arithmetic and Logical Unit
AQUA	Adiabatic QUantum Algorithm
CED	Canny Edge Detection
CP	Controller Processor
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DAG	Direct Acyclic Graph
DMA	Direct Memory Access
DP	Double Precision
DSL	Domain Specific Language
DT	Distance Transform
EDT	Euclidean Distance Transform
extCED	Extended Canny Edge Detection
FPU	Floating Point Unit
GDLS	Globally Distributed Locally Shared
GPU	Graphics Processing Unit
HLSL	High Level Shader Language
HPC	High Performance Computing
HPP	Heterogeneous Parallel Platform
ISL	Iterative Stencil Loops
MIMD	Multiple Instruction Multiple Data
MIS	Minimally Invasive Surgery
MP	Multiprocessor
MPI	Message Passing Interface
NUMA	Non Uniform Memory Access
NOC	Network On-Chip
OpenCL	Open Computing Language
PCI	Peripheral Component Interconnect
PDE	Partial Derivative Equation
PE	Processing Element
PET	Positron Emission Tomography

RO Cache	Read Only Cache
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SP	Streaming Processor
TBB	Threading Building Block
UTS	Unbalanced Tree Search

CHAPTER 1 INTRODUCTION

1.1 Context : Programming Parallel Platforms

Modern applications, such as image processing and computer vision, are becoming more and more complex (Djabelkhir and Seznec, 2003). This complexity implies a large amount of processed data and high computation loads, under very tight constraints : more constraints are added to enforce high quality and accuracy such as real-time execution and power consumption awareness. Therefore, optimization techniques and high performance hardware platforms are required. Considering the domains of image processing and computer vision, it is safe to say that they present many opportunities for parallelism, that can be exploited in parallel platforms.

With the technology improvement for both hardware and software, parallel platforms started a new computing era and they are here to stay (Jerraya and Wolf, 2004). Parallel platforms may be found in High Performance Computers (HPC) or embedded computers. Recently, both HPC and embedded computers are moving toward heterogeneous computing. They are employing both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to achieve the highest performance. For instance, the supercomputer code-named Titan uses almost 300,000 CPU cores and up to 18,000 GPU cards (Cook, 2012). Programming efficiently for parallel platforms bring new opportunities but also several challenges (Williams et al., 2009). Therefore, industry needs help from the research community to succeed in its recent dramatic shift to parallel computing.

Currently, two main approaches are used for parallel programming : (1) writing new parallel code from scratch, which is an effective way to accelerate applications but time consuming or (2) mapping existing sequential algorithms to parallel architectures to gain speedup. Since, usually, existing algorithms are initially described as high-level sequential code (such as MATLAB and C/C++ code), currently the approach (2) presents a substantially less expensive alternative, and does not require to retrain algorithm developers. This requires the parallelization of sequential code, and the exploitation of set of parallel programming models. This approach is not trivial, as many different hardware target architectures are available : the target platform can be heterogeneous and involve diverse computational architectures and different memory hierarchies. Moreover, for new parallel implementations, the programmer starts by extracting the parallelization opportunities of the application and then expresses

these opportunities by employing a number of parallelization strategies. These parallelization strategies should be suitable for both application programming features and the architecture specificity.

1.2 Parallel Programming - Challenges

Parallel programing presents several major challenges. These challenges are equally present whether one programs on a many-core GPU or on a multicore CPU. Unfortunately, there is little compiler technology that can help programmers to meet these challenges today. Three of the main challenges are :

- Finding the best platform providing the required acceleration – although performance studies of individual platforms exists, they have been so far limited in scope and designers still face daunting challenges in selecting a new appropriate parallel platform in order to accelerate their application (Calandra et al., 2013).
- Selecting the best parallelization strategy – using a proper parallelization strategy is crucial in order to achieve the optimal performance, such that the calculation takes the shortest possible time. It is very difficult to give a universal method for finding the optimal strategy, since it depends not only on the platform, but also on the application itself (Kegel et al., 2011a). Designers need today some guidelines in their search for the parallelization strategy.
- Performance tuning – performance is the primary goal behind most parallel programming efforts. In order to achieve high performance, designers need to tune several parameters (data and task decomposition granularity and threads and data mapping on the target hardware resources) and explore a huge space of solutions. Currently, the performance tuning can take significant time and effort (Kamil, 2013).

1.3 Objectives and Contributions

To alleviate the challenges described in the previous section, the global objective of this thesis is to propose a new solution helping designers to efficiently program complex applications on modern parallel architectures. The specific objectives are :

- Integrating, in the programming stage, some key aspects such as the parallelization strategies : parallelism models, parallelization granularity and programming models.
- Accelerating the programming stage and improving the exploration of the solutions space.

The main contributions of this project are :

1. The evaluation of the efficiency of several target parallel platforms to speedup compute intensive applications
2. Performance Evaluation of parallelization and implementation strategies on multicore CPUs and manycore GPUs.
3. The definition and implementation of a new performance tuning framework for heterogeneous parallel platforms.

These contributions are briefly described in the next three sections.

1.3.1 Evaluation of the Efficiency of Several Target Parallel Platforms to Speedup Compute Intensive Applications

The performance evaluation is performed through three fundamental algorithms used in image processing and computer vision : (1) Canny Edge Detection (CED), (2) Shape Refining and (3) Distance Transform (DT). These algorithms are compute intensive and highly data parallel. We developed for each algorithm a number of parallel implementations targeting a wide range of architectures covering both HPC and embedded parallel platforms. A number of optimizations is investigated depending on the applications and the target architecture. We show the impact of such optimizations on performance via a large number of experiments.

1.3.2 Performance Evaluation of Parallelization and Implementation Strategies on Multicore CPUs and Manycore GPUs

Since applications are increasing in complexity and present a wide variety of parallel features, it becomes difficult to decide which parallelization strategy is suitable for a given platform to reach peak performance. We propose a comprehensive performance evaluation of a large variety of parallelization and implementation strategies for different parallel structures on two most common parallel platforms : a multicore CPU (Dual AMD Opteron 8-core CPU) and a manycore GPU (NVIDIA GTX 480 GPU). Moreover, we consider a wide spectrum of parallel programming models : OpenMP and TBB targeting multicore CPU, and CUDA and OpenCL targeting manycore GPUs. We also aim at determining guidelines for the efficient parallelization strategies of common application classes.

1.3.3 A New Performance Tuning Framework for Heterogeneous Parallel Platforms

In order to implement efficiently compute intensive applications on heterogeneous parallel platforms, a number of parameters have to be considered : data and task decomposition granularity and threads and data mapping on the target hardware resources.
The knowledge acquired during the two first contributions help us to efficiently delimit the exploration space and to define precisely the influential parameters on performance such as the data and task granularity level and the data access pattern.
As contribution, we propose a performance tuning framework for stencil computation targeting heterogeneous parallel platforms. The emphasis is put on platforms that follow the host-device architectural model where the host corresponds to a multicore CPU and the device corresponds to a manycore GPU. This framework guides the developer to select (i) the best task and data granularity also known as, respectively, fusion and tiling, and (ii) the best thread block configuration to fit the problem size.

1.3.4 Document Plan

This thesis is structured in five chapters. In Chapter 1, we presented the context and challenges of our research work and we introduced the objectives and contributions of this thesis. Chapter 2 presents the basic concepts concerning the parallel programming of complex applications on modern parallel platforms. Chapter 3 introduces the evaluation of the parallel platforms for accelerating compute intensive applications. Chapter 4 describes our performance evaluation of the parallel programming and implementation strategies. In Chapter 5, we present a new framework enabling the performance tuning for complex applications running on heterogeneous parallel platforms. Finally, in Chapter 6, we present the conclusions and the perspectives of our research work.

CHAPTER 2 BACKGROUND AND BASIC CONCEPTS

The main goal of this chapter is to introduce the parallel programming world to the reader. In more details, Section 2.1 presents the context of the use of parallel programming to accelerate current applications and in particular image processing and computer vision. Section 2.2 describes basic parallel programming concepts, which are necessary to understand an efficient parallel implementation. Section 2.3 lists the main basic multiprocessor architectures used as target platforms for current parallel applications. Some examples of these platforms are given in Section 2.4. Section 2.5 classifies parallelization strategies according to the parallelization granularity and the types of parallelism. Finally, Section 2.6 gives an overview of existing parallel programming models followed by Section 2.7 which lists some examples of parallel programming models used for both Multicore CPUs and Manycore GPUs.

2.1 Context

Modern applications are becoming more and more complex (Djabelkhir and Seznec, 2003) due to the increasing need of both high quality and accurate computation results. As a consequence, this complexity is translated into a large amount of processed data and high computation loads which require optimized algorithms and high performance hardware platforms to run these algorithms within a reasonable amount of time. More constraints could be added to high quality and accuracy such as real-time execution and power consumption awareness. Image processing and computer vision the are typical domains which follow this trend. The main particularity of these two domains is the presence of high amounts of parallelism which makes the parallel platforms an appropriate hardware platform to satisfy the above mentioned constraints.

Parallel platforms are showing an extraordinary expansion, especially with the technology improvement for both hardware and software, making them a new trend for the computer science domain. parallel platforms may be found in High Performance Computers (HPC) or embedded computers. Recently, both HPC and embedded computers are moving toward heterogeneous computing. They are employing both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to achieve the highest performance. As an example, the supercomputer code-named Titan uses almost 300,000 CPU cores and up to 18,000 GPU cards (Cook, 2012).

In order to program parallel platforms, there are two approaches for parallel programming : (1) writing a new parallel code from scratch, which is an effective way to accelerate applications but time consuming, or (2) mapping existing sequential algorithms to parallel architectures to gain speedup. As existing algorithms are initially described as high-level sequential code (such as MATLAB and C/C++ code), currently the approach (2) presents a substantially less expensive alternative, and does not require to retrain algorithm developers. This requires the parallelization of sequential code, and requires involving a convenient set of parallel programming models. This step is not trivial, as many different hardware target architectures are available : the target platform can be heterogeneous and involve diverse computational architectures and different memory hierarchies.

In parallel programming, the programmer's role is to extract the parallelization opportunities of the application and express such opportunities by employing a number of parallelization strategies. These parallelization strategies should be suitable for both application programming features and the architecture particularity.

2.2 Basic Concepts of Parallel Programming

In this section, we present some of the basic concepts of parallel programming that are underlined in parallel programming.

Concurrency is the decomposition of a program into parallel running sections. The main idea is to look at the task and data dependencies and divide the sections of the program into independent sets.

Locality is to maintain processed data close to the processing units. We can distinguish two kinds of locality : (1) temporal locality - data previously accessed will likely be accessed again and (2) spatial locality - data that is close to the lately accessed data will likely be accessed in the future. Data locality is managed by the cache hierarchy present in multicore CPUs and even in GPUs.

Parallel Patterns are abstracted models that describe the implementation of parallel programs according to some parallelization features. Some of the common parallel patterns are listed below :

- **Loop-based Pattern** is the implementation of a parallel program at the loop-level where each iteration or a group of iterations could be executed independently. The loop-based pattern may be embraced to implement a large number of applications, in

particular in image processing applications.

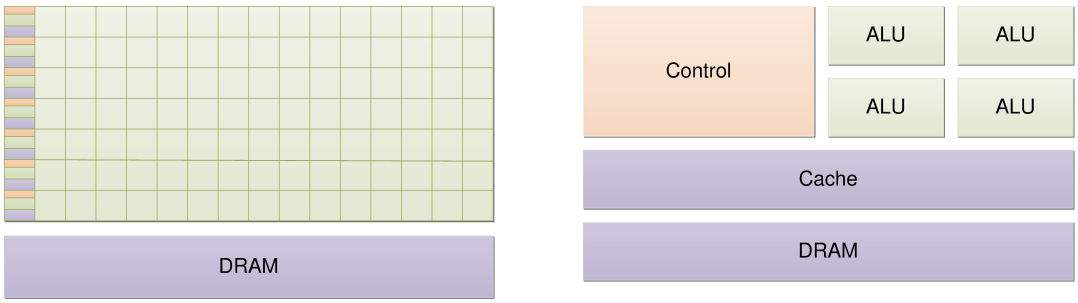
- **Fork/join Pattern** describes the implementation of programs that are composed of a sequence of sequential and parallel sections. At the beginning, only one master thread is running and, when a parallel section is encountered, it creates a group of threads. This action is known as **Fork**. Once the parallel section is finished, groups of threads are joined at a synchronization point. This action is known as **Join**. One of the common programming models suitable for this pattern is OpenMP.
- **Tiling or Domain Decomposition Pattern** consists in splitting the data space into smaller data parts named **tiles**. Each tile is processed by one or a group of threads. At the end, the resultant tiles build the final result.
- **Divide and Conquer Pattern** consists in dividing a large problem into small sub-problems where each could be solved separately. The final result is the sum of the partial sub-problem results. This pattern is often used to implement recursive programs.

2.3 Hardware Parallel Architectures

We can classify parallel architectures according to two aspects : (1) their computation models or (2) their memory organizations.

we distinguish mainly two classes of parallel systems according to their computation model :

- **SIMD Systems :** SIMD stands for Single Instruction Multiple Data. The SIMD architecture is arithmetic units-centric with the cost of a simple control unit (see Figure 2.1(a)). This makes SIMD more suitable for data-oriented applications. By removing control logic complexity and adding more ALUs, SIMD is able to run at high clock rate with low power consumption. GPUs adopts more a relaxed computation model than SIMD. It implements the Single Instruction Multiple Threads (SIMT) model, which is a thread-centric model that gives more flexibility to implement complex data-parallel applications.
- **MIMD Systems :** MIMD stands for Multiple Instruction Multiple Data. MIMD systems integrate a complex control unit for each physical core (see Figure 2.1(b)). This offers the ability to fetch and decode different instructions on each core at the same time. MIMD systems offer more flexibility than SIMD systems to parallelize a wide range of applications. However, they usually suffer from low degree of concurrency compared to SIMD systems. As example of MIMD systems, we name the mainstream multicore CPUs and STHORM, the ST embedded platform.



(a) SIMD : GPU Architecture (b) MIMD : CPU Architecture

Figure 2.1 SIMD : GPU Architecture vs. MIMD : CPU Architecture (Cheng et al., 2014)

We can distinguish essentially three main system architectures according the memory organization : (1) shared memory systems, (2) distributed memory systems, and (3) distributed shared memory systems (Protic et al., 1998).

- **Shared Memory Systems :** In shared memory systems, one single memory space is shared between processors. The main advantages of this architecture are : (1) communications are implicit and efficient, (2) easy to convert sequential program to parallel version, (3) existence of parallel programming models and tools suitable for this architecture. However, the limitations of this architecture are : (1) Not scalable for massive parallel platforms limited by memory contention (Gordon et al., 2006) and (2) need of data coherency and race condition avoidance mechanisms. This form of architecture is adopted in the existing multicore architectures.
- **Distributed Memory Systems :** In distributed memory systems, each processor has its own memory. The advantages of this architecture are : (1) scalability for massive multiprocessor systems and (2) exclusion of data race conditions. However, the limitations of this architecture are : (1) the communications overhead, and (2) the difficulty to balance the work load on Processing Elements (PEs) (Kumar et al., 1994).
- **Distributed Shared Memory Systems :** Distributed shared memory systems are also known as a Globally Distributed Locally Shared (GDLS) systems. GLDS System is composed of clusters where each cluster has its own memory space and each cluster is composed of cores that share a local memory space. The Advantages of this architecture are : (1) the flexibility to organize the shared memory, (2) The scalability to a large number of cores, since the memory contention is avoided by the multilevel memory hierarchy, and (3) transparent programming (shared). We can find this kind of architectures in multi-sockets or multi-nodes multicore CPUs systems to build a NUMA

architecture. More often, we find GDLS systems in existing manycore systems and in particular in current GPUs.

2.4 Examples of Parallel Platforms

We may find basically two main parallel platforms as most used target platforms for parallel applications : (1) Multicore CPU-based Platform and (2) Manycore GPU-based Platform.

2.4.1 Multicore CPU-based Platforms

Multicore CPUs refer to general purpose processors integrating multiple cores in the same die. In general, these cores are identical and they are based on x86 architecture with complex control units which allow large coverage of types of parallelism. Typically CPU cores may handle a maximum of two concurrent threads due to the limited size of the register file. Each context switch between threads introduces additional overhead unlike GPUs which offers a nearly zero overhead through a large register file size. Multicore CPUs are cache-based architectures since they integrate a multi-level cache hierarchy (see Figure 3.8). The memory model multicore architectures follow is the Shared memory model where all cores are connected to the same memory and share the same address space (Ourselin et al., 2002). This particular feature offers more data locality but may yield memory contention due to the simultaneous access to the same address space. This fact limits the number of cores per node. One solution to expand the number of cores is to group a small number of cores into nodes to form a Non Uniform Memory Access (NUMA) system. Cores from the same node share the same address space with low latency access and access to remote node memory with high latency. The programming challenge of such platforms is to ensure : both thread affinity (to keep thread assigned to the same core to avoid thread migration and data migration overhead) and data affinity (to keep data close to the processing thread). Still, multicore CPUs are limited to the order of tens of cores which limits the degree of concurrency.

2.4.2 Manycore GPU-based Platforms

The application of GPUs is no longer restricted to graphics applications. The GPU Platform found its way to a large variety of general purpose computing tasks. In the last years, many algorithms were implemented on graphics hardware (Navarro et al., 2014). In many domains, GPU based hardware architectures are considered among the cost-effective parallel architectures to implement data-oriented applications. Image processing and computer vision applications are taking advantage of performance features offered by GPUs such as high

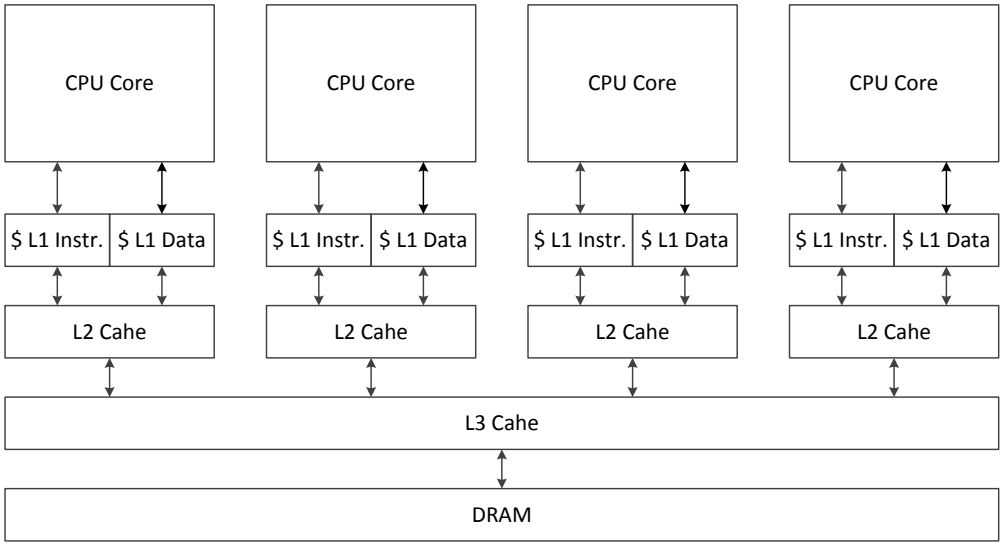


Figure 2.2 Typical Multicore CPU Architecture (Cook, 2012)

degree of data-parallelism, floating point units reaching 3 TFLOPS (compared to only 200 GFLOP for CPUs) and a high memory bandwidth of around 320 GB/s (versus 20 GB/s for CPU, see Figure 2.3).

The evolution of these architectures took a relatively, long time to be considered as a general purpose architecture. This is due to the absence of effective and simple programming languages and tools. Mainly, GPUs were used in specific applications such as games and image rendering. The existing programming languages at that time were essentially shader programming languages such as Cg (Mark et al., 2003), High Level Shader Language (HLSL) (Peeper and Mitchell, 2003)and OpenGL (Rost et al., 2009). All of these programming languages are based on the image rendering process. So, they are compiled into vertex shader and fragment shader to produce the image described by the program. Although, these high level programming languages abstract the capabilities of the target GPU and allow the programmer to write GPU programs in a more familiar C-like programming language, they do not stay far enough from their origins as languages designed to shade polygons.

With the venue of general purpose GPU (GPGPU) ecosystems (architecture, language, runtime, libraries and tools), programs may conceptually have nothing to do with drawing geometric primitives and fetching textures, unlike with the shading languages. Rather, GPGPU programs are often best described as memory and math operations, concepts which are more

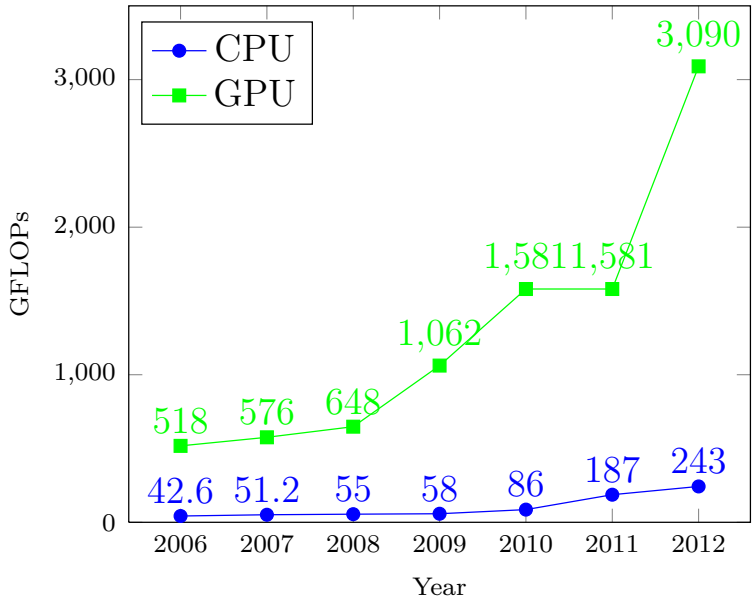


Figure 2.3 CPU and GPU Peak Performance in GFLOPs (Cook, 2012)

familiar to CPU programmers. To make this possible, some work was done such as the Brook programming language which extends ANSI C with concepts from data stream programming (Buck et al., 2004), Scout programming language designed for scientific visualization (McCormick et al., 2004) and Glift template library which provides a generic template library for a wide range of GPU data structures (Lefohn et al., 2006). In 2007, NVIDIA brought GPUs to the GPGPU programming domain by developing a developer friendly programming model called Compute Unified Device Architecture (CUDA) (Nvidia, 2007) considered as a most efficient parallel programming model for NVIDIA GPUs. Similarly AMD designed GPGPU architectures and contributed to the development of the Open Computing Language (OpenCL) (Stone et al., 2010) to program their GPUs. Since then, other parallel programming models are developed for GPUs such as DirectCompute (Ni, 2009), and OpenACC (Group et al., 2011).

Both NVIDIA and AMD GPUs are composed of a number of multiprocessors (MP) named respectively by NVIDIA as Streaming Multiprocessors (SM) and by AMD as Compute Units (CU) (see Figure 2.4). Each MP integrates a large number of streaming processors (SPs) where in NVIDIA GPUs, each SM is a basic core integrating an integer arithmetic and logical unit (ALU), and a floating point unit (FPU). On the other side, AMD GPUs follow the SIMD fashion where cores integrate ALUs and are grouped into SIMD engines. Each MP is equipped with a number for schedulers and dispatch units responsible of scheduling threads

on the available cores and allocating the appropriate hardware resources. Besides processing units, each MP integrates a relatively small fast local memory that is accessed only by the SPs of the same MP via a network on-chip (NOC). In NVIDIA GPUs, this memory may be configured in part as an L1 cache and the rest as explicitly managed memory (scratchpad memory). In AMD GPUs, the L1 cache and the scratchpad memory are integrated as separate memories. In addition to memories, a huge number of registers is integrated in each MP to allow the support of a large number of threads working together with negligible switch context overhead. MPs from both GPU vendors share an L2 cache and a large high latency memory known as global or device memory.

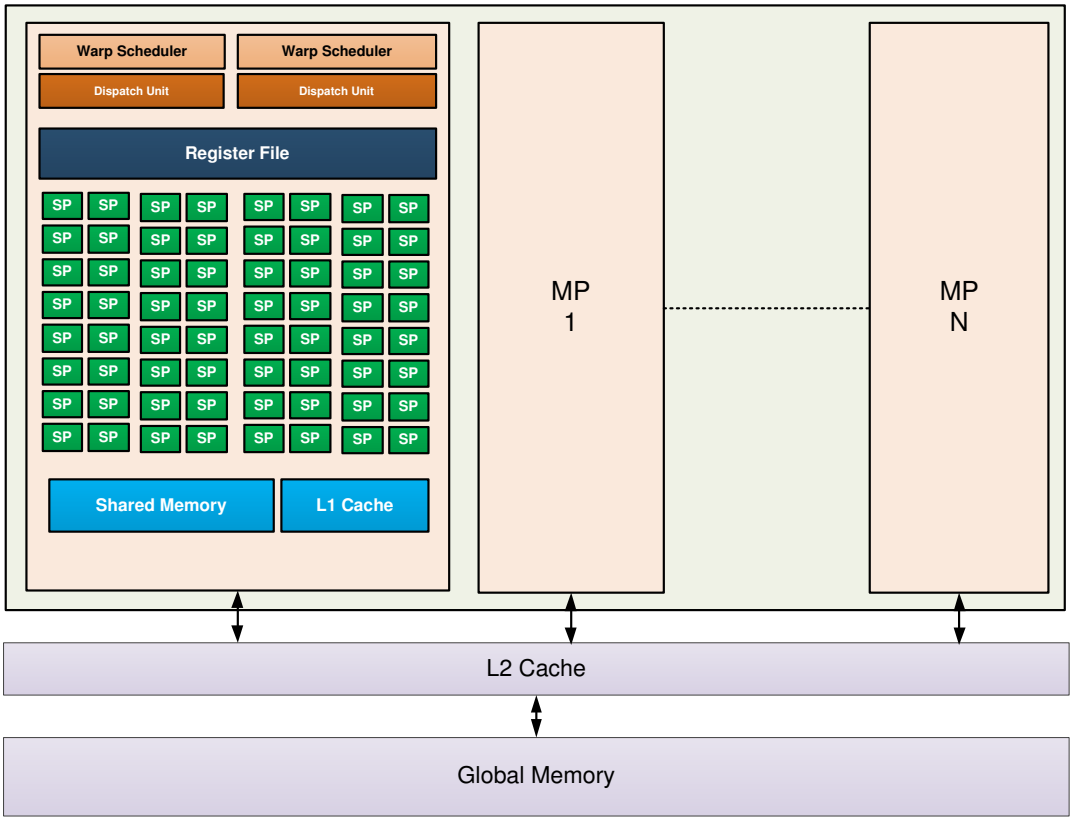


Figure 2.4 Example of GPU Architecture

2.5 Parallelization Strategies

Parallelization Strategies define how to implement parallelization opportunities extracted from an application. In the following, we classify parallelization strategies according the parallelization granularity and types of parallelism.

2.5.1 Parallelization Granularities

Parallelization granularity refers to the size of parallel entities at which we can divide an application. Parallel entity may be tasks or processed data. We can distinguish two levels of parallelization : (1)task-level parallelization and (2) data-level parallelization. For both levels we may define mainly two granularities : (1) fine-grain and (2) coarse-grain.

Task-level Granularity

Task-level granularity is directly related to the program decomposition into independent tasks.

- **Fine-grain tasking** consists in dividing the program in elementary separate tasks and each single task is parallelized apart. This process is called **fission**. The benefit of the fine-grain parallelization strategy is the high reusability since each task may be found in more than one algorithm which is the case of most image processing algorithms. This means that a parallel version of this operation can be reused more than once in different applications without any modification to ensure high portability among different applications. However, this strategy may suffer from (1) the overhead introduced by successive threads launches and (2) from poor temporal data locality.

- **Coarse-grain tasking** consists in packing a sequence of tasks into a macro task. This process is called **fusion**. Each macro task is assigned to a single thread which processes a part of data array. The implementation of this parallelization strategy needs additional programming effort to manage dependencies between neighbors data in order to minimize the synchronization barriers and data communication. When the implementation of the coarse-grain strategy is optimized, runtime performances may be improved by increasing temporal data locality and by avoiding overhead caused by threads launches and data transfers.

Data-level Granularity

Data-level granularity defines the degree of decomposition of initial data into data subsets.

- **Fine-grain Tiling** consists in decomposing the data into small subsets (small tiles in the case of image processing). These small tiles are assigned to small groups of threads.

This strategy exposes a high degree of concurrency and takes advantage of architectures supporting a huge number of threads. In addition, this strategy is usually not constrained by the hardware resources limitations. However, it suffers from a significant overhead in processing replicated data at borders to handle boundary dependencies. The data replication overhead is proportional to the number decomposed data. This strategy also suffers from poor space locality.

- **Coarse-grain Tiling** consists in decomposing data into large data subsets and involving large groups of threads. This strategy reduces the data replication at borders and offers high space data locality. However large tiles may not fit well with available resources, cache or local memory size, which may degrade performance.

2.5.2 Types of Parallelism

Types of parallelism or, also known as parallelism models, define the way to organize independent workflows. We can distinguish three main types of parallelism : (1) data parallelism, (2) task parallelism and (3) pipeline parallelism.

Data Parallelism

Data parallelism refers to work units executing the same operations on a set of data. The data is typically organized into a common structure such as arrays or vectors. Each work unit performs the same operations as other work units but on different elements of the data structure. The first concern of this parallelism type is how to distribute data on work units while keeping them independent. Data parallelism is generally easier to exploit due to the simpler computational model involved (Orlando et al., 2000). However, data parallelism is limited to some application aspects and it cannot exist alone with the increasing complexity of modern applications (Gordon et al., 2006).

Task Parallelism

Task parallelism is known also as functional parallelism or control parallelism. This type of parallelism considers the case when work units are executing on different control flow paths. Work units may execute different operations on either the same data or different data. In task parallelism, work units can be known at the beginning of execution or can be generated at runtime.

Task parallelism could be expressed in the GPU context in two ways. In a multi-GPU environment, each task could be processed by a separate GPU. In a single GPU environment, the task parallelism is expressed as independent kernel queues called respectively streams and command queues in CUDA and OpenCL. kernel queues are executed concurrently where each kernel queue performs its workload in a data-parallel fashion.

Pipeline Parallelism

Pipeline parallelism is also known as temporal parallelism. This type of parallelism is applied to chains of producers and consumers that are directly connected. Each task is divided in a number of successive phases. Each worker processes a phase of a given task. The result of each work unit is delivered to the next for processing. At the same time, the producer work unit starts to process a given phase of a new task. Compared to data parallelism, this approach offers reduced latency, reduced buffering, and good locality. However, this form of pipelining introduces extra synchronization, as producers and consumers must stay tightly coupled in their execution (Gordon et al., 2006).

One form of pipelining parallelism that is often used in parallel programming is overlapping data transfer and processing. This form of pipelining is performed via the Direct Memory Access (DMA) mechanism. In GPU architectures, DMA is used to transfer data between host memory and GPU global memory via PCI.

2.6 Parallel Programming Models

The programming models expose the parallelism capabilities to the programmer while they abstract the underlying architecture away. The programming models have to show certain architecture features such as the parallelism level, the type of parallelism, and the abstraction degree of the components' functions. Parallel programming models are implemented as a set of languages, extensions of existing languages, libraries and tools to map applications on parallel architectures.

Inspired by the classification of the parallel programming models proposed by P. Paulin et al. in (Paulin et al., 2006), we can group the parallel programming models in three main classes : (1) Shared memory programming models, (2) distributed memory programming models and (3) Data streaming parallel programming models.

2.6.1 Shared Memory Programming Model

It is known as the simplest way for parallel programming. It refers to shared memory systems in which the application is organized as a set of processes sharing the same memory. By programming with this model, some form of load balancing is satisfied (Nicolescu and Mosterman, 2009). However, it is necessary to ensure the data coherency which is the greatest challenge of this parallel programming model. The main limitation of the shared-memory programming model is that it does not support heterogeneous systems (Nicolescu and Mosterman, 2009).

2.6.2 Distributed Memory Programming Model

Inspired from the client-server model, the distributed memory parallel programming model is still appropriate for heterogeneous systems and control-oriented applications. The distributed memory parallel programming is scalable with the increasing number of processing elements. Moreover, it is well suited to express the coarse-grained parallelism by offering generalized communication mechanisms. However, these mechanisms demand more effort to fit them to the infrastructure which implies performance overhead.

Message Passing Interface (MPI) (Pacheco, 1997) is considered the de-facto standard for deploying applications on distributed memory systems. We can find several implementations of MPI such as OpenMPI (Graham et al., 2006), MVAPICH Team (2001).

2.6.3 Data Streaming Programming Model

Data streaming parallel programming models are motivated by the architecture evolution which is shown in the graphical processing architectures offered by NVIDIA and AMD. In addition, the emerging need to map aerospace applications, network applications and multimedia applications on such architectures make this parallel programming model more required. Based on its nature suitable for data-oriented parallelism, this programming model helps programmers to reach high performance by reducing the communication overhead. This programming model is implemented by languages such as StreamIt (Thies et al., 2002) and Brook or language extensions such as CUDA or OpenCl.

Data streaming parallel programming is based on two basic concepts :

Streams : contain a set of elements of the same type. Streams can have different lengths and the elements can be simple as float or complex as structure of floats, as

defined in the Brook language. One should not confuse this with stream in the CUDA terminology which defines an execution queue of operations.

Kernels : consist of a loop that processes each element from the input stream. The body of the loop first pops an element from its input stream, performs some computation on that element, and then pushes the results into the output stream.

In the data streaming parallel programming model, a program is represented as a set of autonomous kernels that are fired repeatedly by stream elements in a periodic schedule. The central idea behind stream processing is to organize an application into streams and kernels to expose the inherent locality and concurrency in media-processing applications. The main limitation of this parallel programming model is its poor support for control-oriented computations.

2.7 Examples of Parallel Programming Models for Multicore CPUs and Manycore GPUs

In this section we give an overview of most used parallel programming models for Multicore CPUs and Manycore GPUs.

2.7.1 Examples of Parallel Programming Models for CPUs

We focus in this thesis on two most used high-level parallel programming models in programming multicore CPUs mainly OpenMP and Intel TBB.

OpenMP

OpenMP is a standard shared-memory programming model (Dagum and Menon, 1998). It is designed as an API used to explicitly command multi-threaded shared memory parallelism. OpenMP consists of a set of compiler directives, runtime library routines and environment variables. It targets two languages C/C++ and Fortran. OpenMP was a thread-centric programming model in previous versions but, since OpenMP 3.0, it is also a task-centric programming model to parallelism at the task-level.

The OpenMP model follows a Fork-Join Model (Lee et al., 2009) where all OpenMP programs begin as a single process : the master thread. The master thread executes sequentially until the first parallel region construct - a directive which indicates that the region of the

program following this directive will be processed in parallel by the available threads. Here is an example of a directive used in OpenMP-C : #pragma omp parallel.

The main feature of OpenMP is the ease of use by providing the capability to incrementally parallelize a sequential program. Moreover, it is capable of implementing both coarse-grain and fine-grain parallelism. However, since it is a compiler-based programming model, the performance of applications implemented in OpenMP may change, depending on the compiler used.

TBB

Intel Threading Building Blocks (TBB) (Pheatt, 2008) is a library-based programming model for multi-core CPUs. It provides a higher level of abstraction to express parallelism : the work units in TBB are specified as tasks, and not threads. TBB targets C++ as the programming language and provides some templates to express a number of defined parallel algorithms. TBB has an asset compared to OpenMP by providing thread-safe concurrent containers.

2.7.2 Examples of Parallel Programming Models for GPUs

In order to program GPU-based platforms, specific APIs known as parallel programming models are developed. In this thesis, we focus on the most used programming models for GPU-based platforms : namely **CUDA** and **OpenCL**. Both CUDA and OpenCL are extensions of the C language and implement a particular runtime to manage the computation on a GPU. The two programming models present lots of similarities but differ in portability and performance, depending on the target architecture. While CUDA is running only on NVIDIA GPUs, OpenCL is ported to several heterogeneous platforms including NVIDIA and AMD GPU-based platforms. The program in both programming models is written as a host-side (CPU) program and a device-side (GPU) program. The host-side program is responsible to initialize the device, allocate data in both host and device, exchange data between host and device and launch work on the device. In both programming models, a device-side program is expressed as a compute kernel, which is the elementary execution unit. A compute kernel is basically a C function that expresses the processing assigned to a single thread. In order to accomplish a parallel execution of the same kernel on a collection of data, the kernel is instantiated on a number of threads by means of the programming model-specific syntax.

CUDA and OpenCL adopt the same philosophy for their runtime models. Threads in CUDA

terminology, or work items in OpenCL terminology, are organized in blocks called respectively thread blocks in CUDA and work groups in OpenCL. Threads inside one block are organized in a 1D, 2D or 3D grid (see Figure 2.5). Threads belonging to the same block are assigned to the same MP and share the same portion of the local memory and the synchronization between them is allowed via barriers (see Figure 2.6). Simultaneous accesses of Threads to the local memory have to be memory conflict-free to take advantage of the full local memory bandwidth. The total number of threads is scheduled as execution units called warp in CUDA and wavefront in OpenCL. Threads belonging to the same execution unit have to follow the same execution path to avoid any execution serialization. Both CUDA and OpenCL runtimes provide a mechanism to support the parallelism between different compute kernels or between compute kernels and data-transfer, which is called stream in CUDA and command queue in OpenCL. This mechanism gives an additional flexibility in expressing computations and improving performance by tailoring the appropriate number of threads to the computation, and hiding the access overhead to the data resident in device memory by overlapping computation and memory transfer.

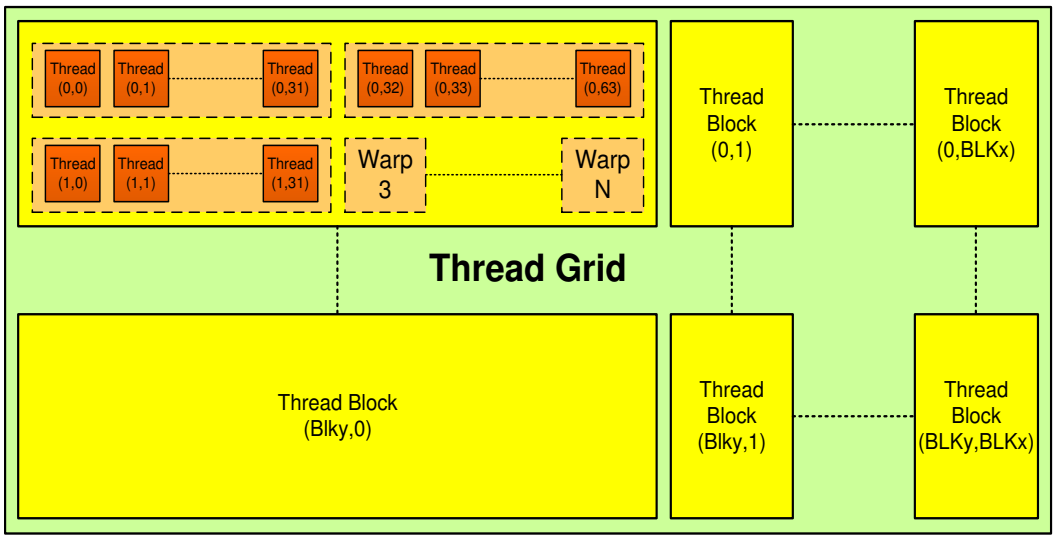


Figure 2.5 Thread Arrangement in GPU Programming Models - Example : CUDA

Recently a new programming model has emerged : OpenACC is considered as an alternative to low-level parallel programming models as CUDA and OpenCL. OpenACC is a directive-based programming model like OpenMP but it targets GPUs with the ability to map data on specific type of memory via simple program annotation. It is a good solution for programmers

looking for functional programs on GPUs with less effort and time. However, this comes with the cost of lower performance compared to low-level parallel programming models.

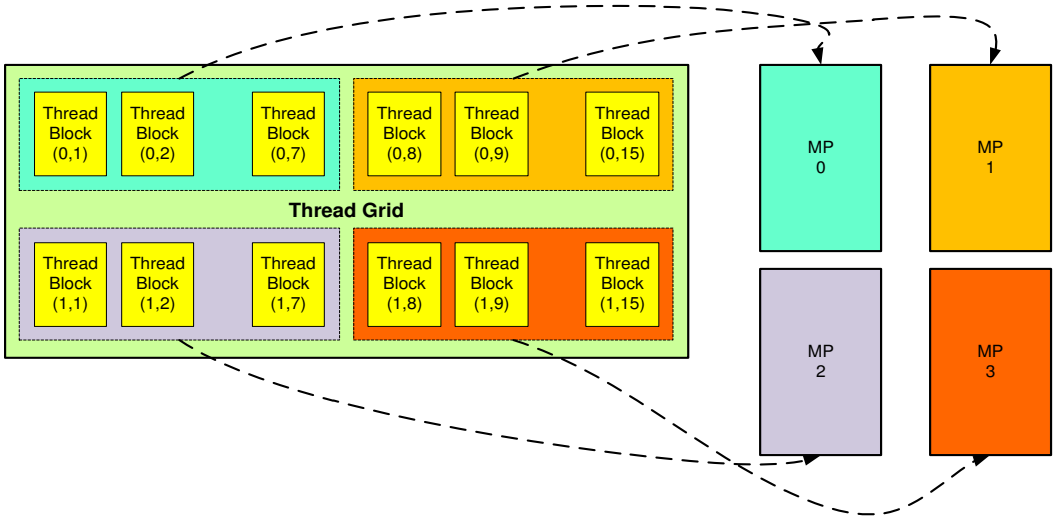


Figure 2.6 Thread Block Mapping on GPU Multiprocessors

2.8 Conclusion

In this chapter, we presented some basic concepts that are necessary to address parallel programming. Also, an overview of the main existing parallel hardware platforms is presented, their programming models and the parallelization strategies that may be employed to parallelize an application. In the next chapter, we go into details for some of the above mentioned concepts through a practical application. To do so, a number of image processing and computer vision applications are implemented on a variety of parallel platforms.

CHAPTER 3 ACCELERATION OF IMAGE PROCESSING AND COMPUTER VISION APPLICATIONS ON PARALLEL PLATFORMS

3.1 Introduction

In this chapter, we evaluate the efficiency of several target parallel platforms to speedup image processing and computer vision applications. The performance evaluation is performed through three fundamental applications used in image processing and computer vision : (1) Canny Edge Detection (CED), (3) Shape Refining : thinning as example and (3) Distance Transform (DT). These applications are compute intensive and highly data parallel. We developed for each application a number of parallel implementations targeting a wide range of architectures including : (1) high performance computers and (2) embedded parallel platforms. A number of parallelization strategies and optimizations techniques are investigated depending on the applications and the target architecture. We show the impact of such optimizations on performance via a large set of experiments.

The chapter is organized as follows. Section 3.2 provides a state-of-art on the parallelization of image processing applications on different hardware platforms. Section 3.3 presents an overview of the used approach to evaluate different parallelization strategies on parallel platforms. Section 3.4 presents the applications context and describes their functionalities. Section 3.5 provides a detailed description for the architecture specifications of the target parallel platforms. Section 3.6 describes in details our approach to parallelize the selected applications on the target platform. Section 3.7 presents the experimental results of different implementations of the studied applications on the target platforms. Finally, Section 3.8 concludes this chapter with a summary of the obtained results.

3.2 Related Work

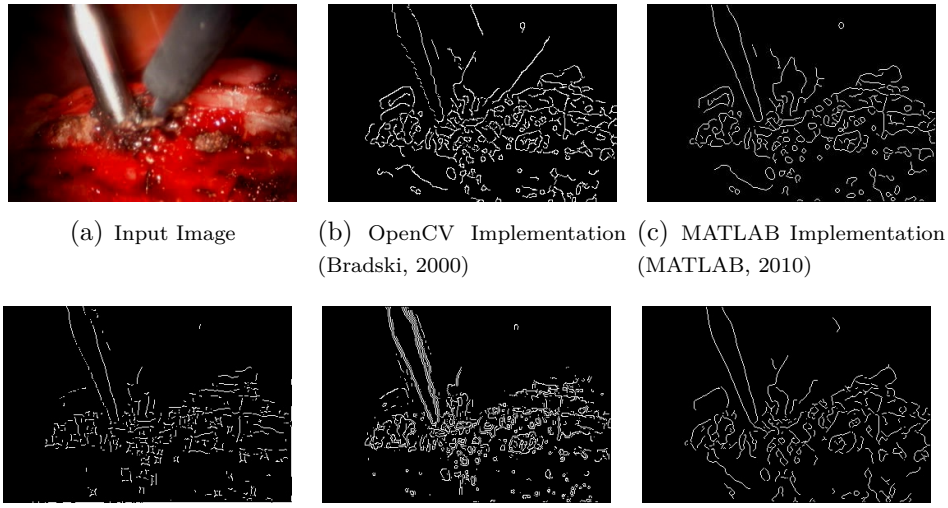
We provide in this section an overview of a number of work accelerating image processing and computer vision applications on parallel target platforms.
Several acceleration efforts of medical imaging processing on GPU are surveyed in (Eklund et al., 2013). The review covers GPU acceleration of commonly used image processing operations in medical imaging (image registration, image segmentation and image denoising). Those operations are filtering, histogram estimation and distance transform.

In Chen et al. (2007a), an improved parallel implementation of the CED application as part of an articulated body tracking application is proposed. The proposed implementation outperforms a non optimized original implementation with respect of speedup (19.91x vs. only 4.16x on 32 threads). These results are obtained on a simulated future multi-core architecture running 32 threads. However, there is no comparison between these two implementations on a real multi-core architecture, while only the original implementation is tested on a real 8 Intel Xeon processors and which reaches only 2.4x as speedup with 8 threads. In Swargha and Rodrigues (2012), three image processing applications (Laplace, Sobel and CED) are implemented using OpenCL on the AMD Radeon HD 6450 GPU which integrates 160 multi-threaded streaming processors (SPs). The parallel implementation of CED on GPU reaches a speedup of 50x compared to a serial implementation on Intel Core i3 CPU running at 2.1 GHz. However, the proposed implementation provides poor detection due to assumptions made toward a reliable implementation on GPU. In Ogawa et al. (2010) and Niu et al. (2011) more accurate parallel CED implementation are proposed using CUDA on NVIDIA GPUs. The former reaches 50x speedup on Tesla C1060 GPU which integrates 240 SPs running at 1.3 GHz. The speedup is achieved compared to Intel Xeon E5540 processor running at 2 GHz. The latter proposes an improved CED on GPU which reaches a speedup between 22x and 55x compared to Intel core 2 Q8400 running at 2.66 GHz. Nevertheless, the proposed CED implementation remains more accurate and most suitable for medical applications.

Other works are presented targeting embedded multiprocessor platforms. In Ensor and Hall (2011), a performance evaluation of a number of heterogeneous multiprocessor architectures involving multi-core CPUs and GPUs for mobile devices is provided. In this context, a GPU shader-based implementation of CED application on OpenGL ES Munshi et al. (2008) is evaluated on several mobile devices : Google Nexus One, iPhone4, Samsung Galaxy S, Nokia N8, HTC Desire HD and Nexus S. The results show that for some devices, the offloading of the edge detection from CPU to the GPU may offer a 50% performance benefit, but for other cases it may be insignificant due to the inter CPU-GPU data transfer overhead. In Patil (2009) and Brethorst et al. (2011), two specific purpose multiprocessor architectures are targeted. The former deals with video games through Cell Be PlayStation 3 architecture which is composed of a host CPU and eight synergetic processing elements. The latter deals with spatial mission through a Tilera Tile 64 processor which is a 64-tiled multi-core architecture. The CED implemented on Cell Be reaches a speedup of 7x compared to Intel Core 2 Duo processor running at 2 GHz and an execution time around 28 ms for 1024x1024 images. The results of the implementation of the CED on Tilera are shown for only 8 cores while it integrates 64 cores. The speedup reached on 8 cores remains poor around 5x and an execution time around seconds.

In Figure 3.1, we provide a quality comparison of detected edges of our implementation versus some of the previously mentioned CED implementations. While OpenCV on CPU and the two GPU reference implementations are fast enough to reach real time execution, they remain inaccurate with non continuous edge lines or noisy extra lines. On the other hand, our implementation, running on low-power SoC at frame rate of 10 frames/s provides accurate smooth line edges.

The most interesting work is reported in (Pallipuram et al., 2012) which provides a rich comparative study of different implementations of Neural Network on two different GPU vendors Nvidia and AMD. Despite the several work on accelerating image processing applications, only few work are performing a deep comparison of the parallelization efficiency of the existing hardware parallel platforms.



(a) Input Image (b) OpenCV Implementation (Bradski, 2000) (c) MATLAB Implementation (MATLAB, 2010)

(d) GPU Implementation (Luo and Duraiswami, 2008) (e) GPU Implementation (Ogawa et al., 2010) (f) Our Implementation

Figure 3.1 Smoothing Edges Comparison : Output Images of Different CED Implementations

3.3 Evaluation Approach Overview

In this section, we present an overview of our approach to evaluate the efficiency of a large set of parallel platforms to run parallel applications (see Figure 3.2). This approach takes the studied applications as input and preforms the following three steps :

1. Analyze each application complexity and available parallelization opportunities,
2. Select appropriate parallelization strategies depending on the target platform characteristics,

3. Implement the parallel versions using each parallelization strategy on target platforms and evaluate the parallelization efficiency.

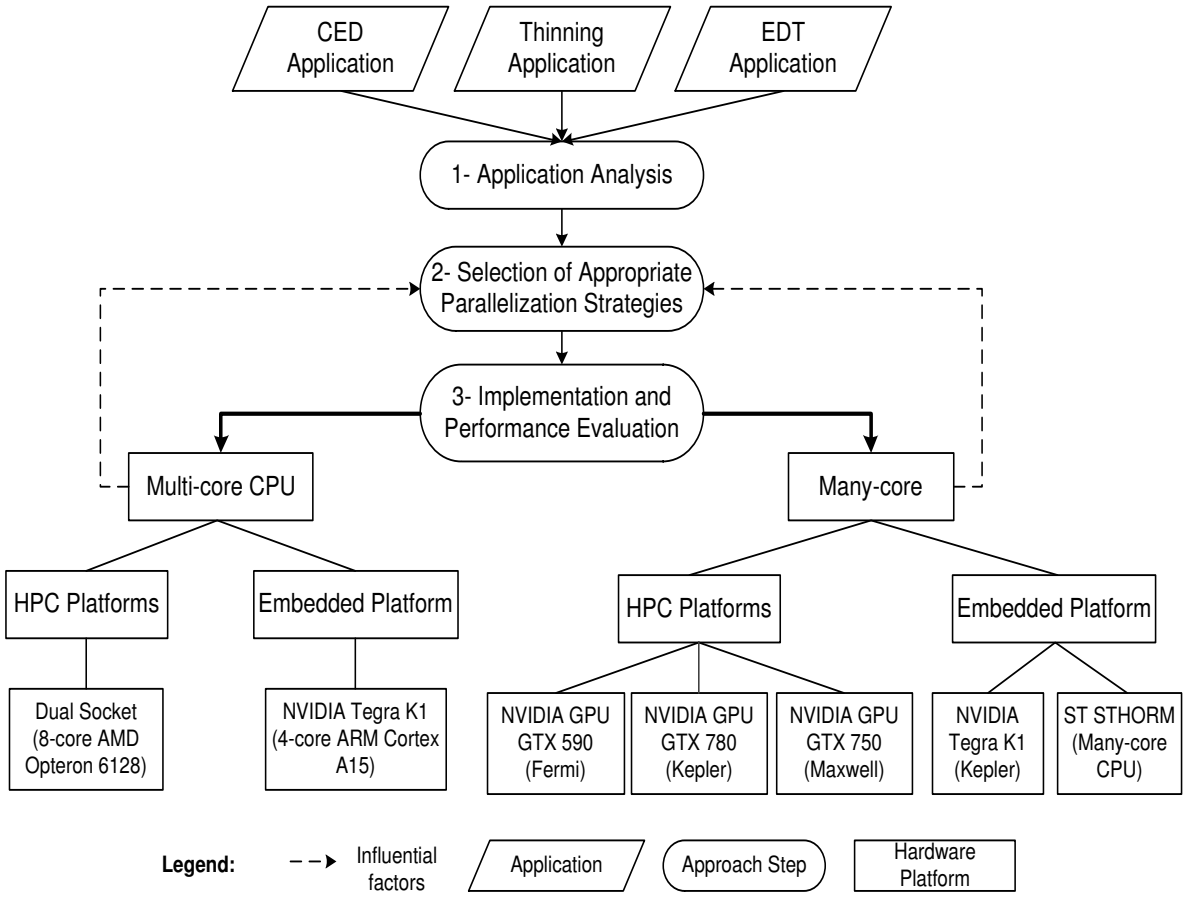


Figure 3.2 Performance Evaluation Approach

3.4 Case Study Applications

As a case study, we target the augmented reality system for Minimally Invasive Surgery (MIS) used for spinal surgery. MIS imposes high difficulty to the surgeon since he has to handle surgical instruments relying on a narrow field of view captured from an endoscope (monocular camera introduced in the patient body via a small hole). An augmented reality system based on two processing stages is the solution to get over the loss of depth perception and the narrow field of view. The first stage consists of a real-time detection of surgical instruments and their positions in the captured video. The second stage consists of mapping the captured

positions on the preoperative 3D spinal model to build an augmented reality scene (see Figure. 3.3). In this thesis, we focus on the processing involved in the first stage (see Figure. 3.4).

As a starting point, we use the application developed by (Windisch et al., 2005). The application is developed in MATLAB (MATLAB, 2010) where all the functional aspects are met. The main issue of the developed application is the long execution time which makes it not suitable for practical clinical use. The challenge is how to speedup such application and maintain high accurate instrument detection. First, we identified the main applications involved in first stage which are : (1) Canny Edge Detection (CED), (3) Shape Refining : thinning as an example and (3) Euclidean Distance Transform (EDT). We used these applications as a case study to evaluate the parallel computation capability of several hardware parallel platforms. In the following, we detail the parallelized applications.

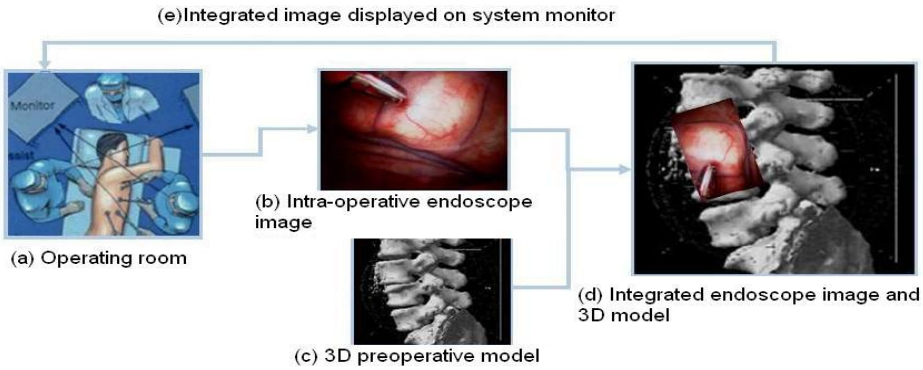


Figure 3.3 Augmented Reality System for spinal MIS (Windisch et al., 2005)

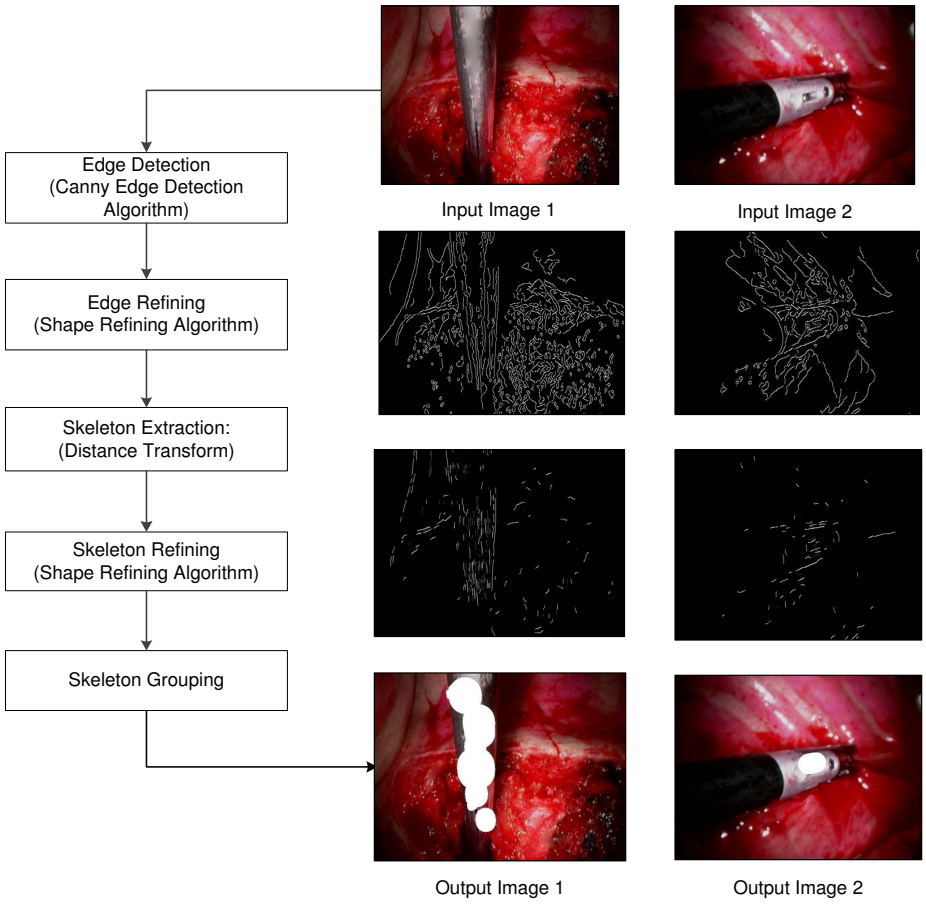


Figure 3.4 Instrument Detection Stages

3.4.1 Canny Edge Detection CED)

The CED (Canny, 1986) is used as a fundamental preprocessing step in several computer vision applications. CED is a multi-stage application (see Figure. 3.5) composed of the following stages :

1. **Gaussian Blur :** it is an example of noise suppression filter based on linear algebra operations. The main operation consists of calculating an output image pixel as the result of a convolution of an input image pixel and its neighborhood with a 2D filter also called *convolution mask* (Sonka et al., 2014). The Gaussian Blur is known to be a computationally costly filter and this cost is proportional to the image and the filter size. However, this cost is considerably reduced since the Gaussian filter has the characteristics of separability. By separating the 2D filter convolution into two successive 1D convolutions, the number of operations is reduced from MxNxHxH to

2xMxNxH where M,N and H are respectively the height, the width of the image and the width of the filter.

2. **Gradient Calculation :** it consists of calculating the derivatives of the image according to the 2D space. This is reduced to calculate the pixels' intensity variation according the horizontal (X) and the vertical (Y) directions by convolving the input image in each direction with a 2D gradient filter. Same as Gaussian Blur convolution, each directional gradient is separated into two successive 1D convolution with 1D gradient filters.
3. **Magnitude Calculation :** it collects the two directional gradients and calculates the magnitude of the gradients for each pixel.
4. **Non Maximum Suppression :** based on the gradient values of each pixel and the directional gradient, this stage groups the pixels into two groups : (1) weak edge pixels (those pixels with gradient values exceeding a low threshold) and (2) strong edge pixels (those pixel with gradient values exceeding a high threshold). A weak edge pixel is not considered as a final edge pixel if it is not connected to a strong edge pixel. The role of the next stage is to keep only real edge pixels.
5. **Edge pixels Connection :** it links together the pixels belonging to the same edge. It pulls one pixel at a time from the strong edge pixels group and searches if any of the neighboring pixels that belong to the weak edge pixels could be connected to it.

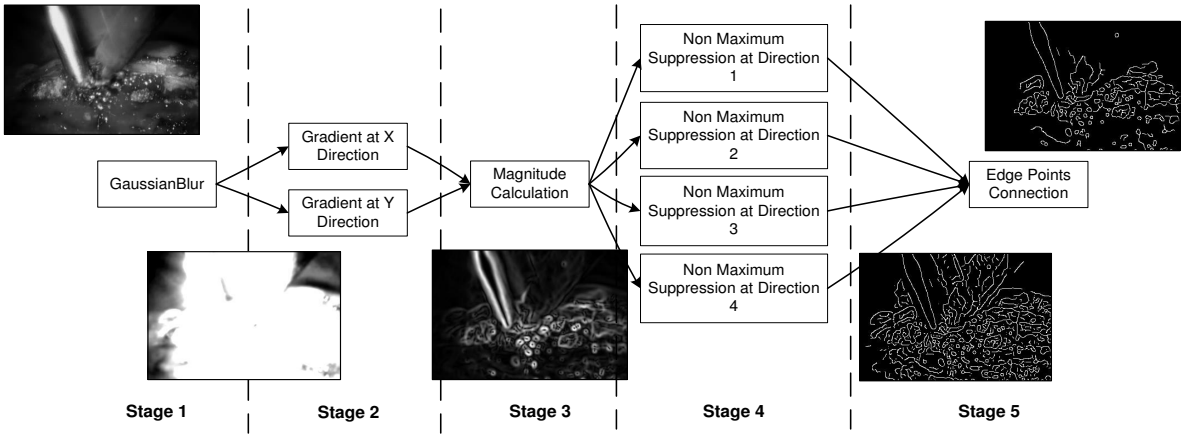


Figure 3.5 Canny Edge Detection Stages

3.4.2 Shape Refining : Thinning

Since the studied application targets medical imaging that requires high accurate detection, another stage to CED that is responsible of refining the edges' shapes is required. This stage involves a well known image processing operation, the thinning morphological operation. The thinning stage consists of performing a number of successive morphological operations until there is no more possible thinning to be done for the output image. At the end of this stage, we obtain the final set of thin edges. This processing is an iterative processing where the number of iterations is unpredictable at compile time and it is image content dependent. By adding this stage to the CED application, we provide high accurate edges but with a price of more complex operations and heavier computations. So to accelerate this application, additional effort in the parallelization task is needed.

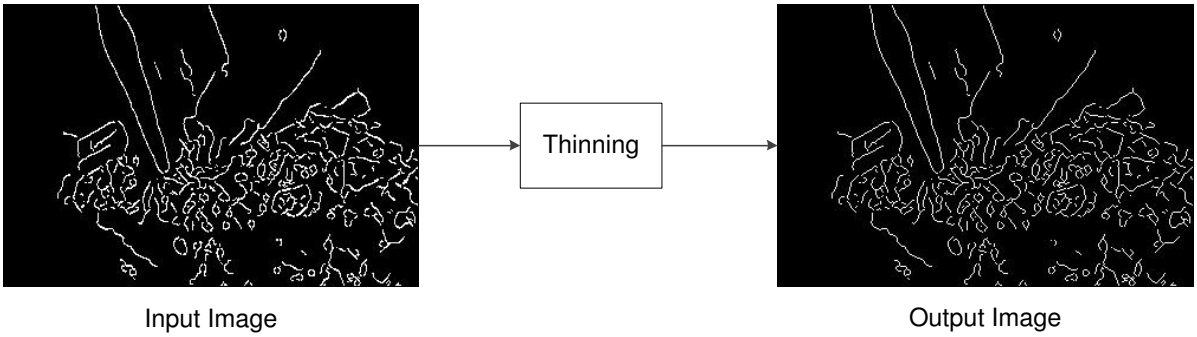


Figure 3.6 Thinning Application

3.4.3 2D Euclidean Distance Transform

The Distance Transform (DT) is a fundamental application used in many computer vision applications. DT is applied in shape analysis, pattern recognition, and computational geometry (Fabbri et al., 2008). The DT maps each image pixel into its smallest distance to regions of interest (Rosenfeld and Pfaltz, 1966). DT is applied to a binary image where a white pixel represents the background and the black pixel represent the object also called the foreground. The image resulting of DT is called also distance map and it is represented by pixels where their intensity is proportional to the computed distance (see Figure. 3.7). Many forms of distance calculation could be used in DT namely city block, chessboard and Euclidean distances (EDT). The latter is known to be the most accurate one but the most computationally expensive. Several applications of DT are listed in (Fabbri et al., 2008) such as :

- Separation of overlapping objects,
- Computation of geometrical representations : skeletonization,
- Robot navigation : obstacles avoidance,
- Shape matching.

A basic example of EDT implementation is the Brute-Force EDT. This application provides the minimum distance between a white and a black pixel.

In this work, we implemented a modified version of Brute-Force EDT. The proposed version is adapted to a specific problem which consists of determining the distance of neighboring pixels to a detected edge. We restrict our EDT computation to only neighbour pixels that belong to a limited region. This limit is set depending on the application. In our case, we consider only pixels that belong to a disk of a preset radius. This application serves as a case study that shows a particular programming feature where memory intensive access conditional processing and work load unbalancing are involved. Consequently, the efficiency of parallel implementation of this application on multicore CPUs and on local store-based manycore platform may be affected. The main stage of the implemented algorithm are :

1. Initialize the distance of every white pixel to the maximum distance value,
2. For each pixel belonging to the edge, map the precalculated distance grid on the neighbor region centered to that pixel,
3. For each neighbor, calculate the distance to the edge pixel. If the new distance is smaller than the current distance, update the neighbor's current distance with the new one.

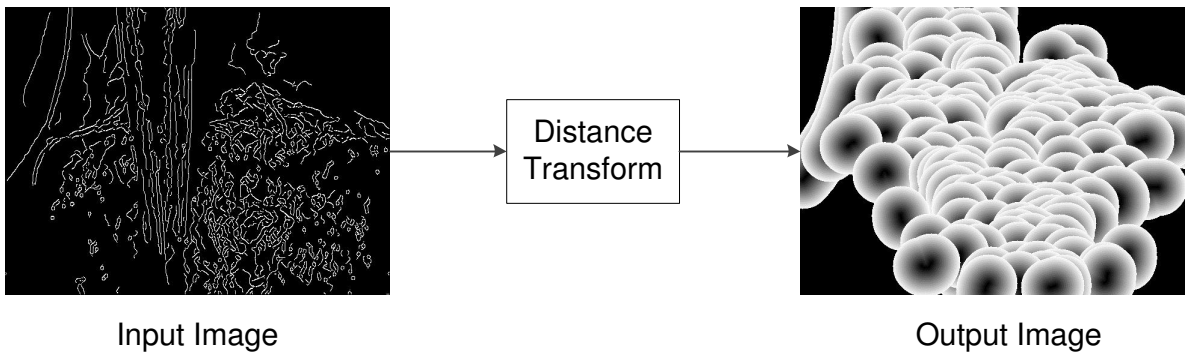


Figure 3.7 Distance Transform Application

3.5 Target Parallel Hardware Platforms

In this section, we give a detailed architectural overview of the parallel hardware platforms used to implement the above described applications. The target platforms cover both HPC and embedded computers.

3.5.1 HPC CPU-Based Platform

As a multicore CPU platform, we use a two AMD Opteron 6128 processors (see Figure. 3.8). Each AMD Opteron processor is composed of 8 cores working at 2 GHz. For each processor, the cache memory is distributed as follows : 8 x 64 KB Data L1, 8 x 64 KB Instruction L1, 8 x 512 KB L2, and 10 MB shared L3. The main memory is distributed as 6 GB DDR3 memory 1333 MHz for each processor. As parallel programming model, we use OpenMP to implement the parallelized studied applications.

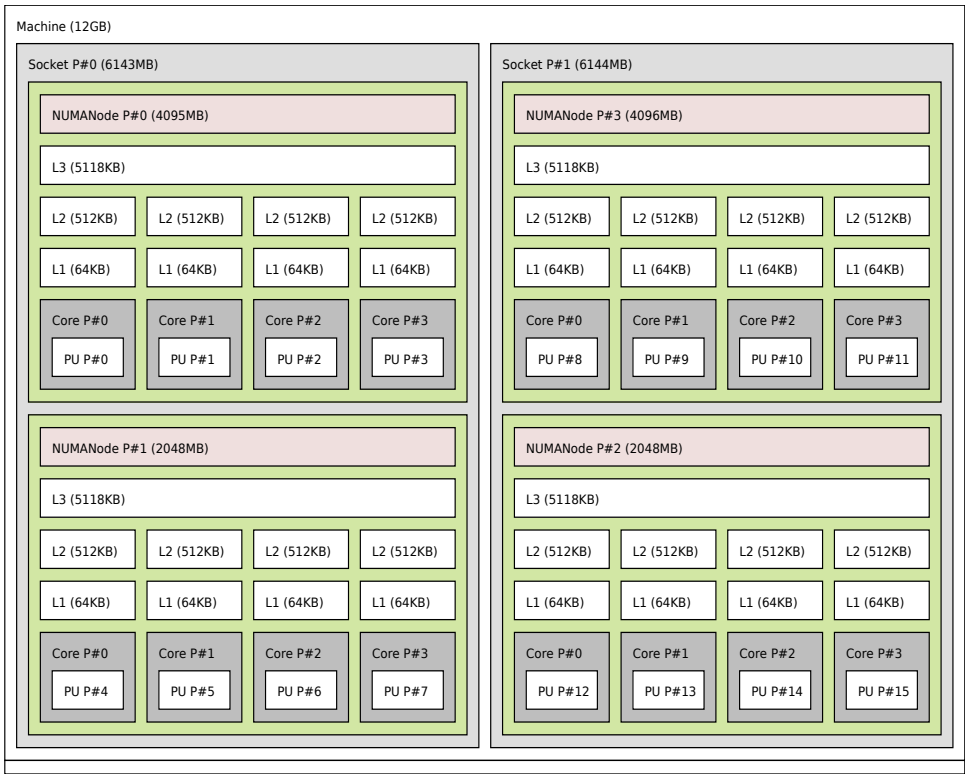


Figure 3.8 Multicore CPU Architecture

3.5.2 HPC GPU-Based Platforms

We study different families of Nvidia GPUs. These GPUs integrate a complex memory hierarchy and a huge number of processing elements that deliver enormous computing power. To program these GPU, we use CUDA as parallel programming model. This section presents the relevant architecture aspects for each platform exploited in our project.

NVIDIA Fermi GPU

The NVIDIA Fermi GPU codenamed GF100 (NVIDIA, 2009) is the first NVIDIA GPU which integrates an L1 cache memory alongside with the shared memory. For our experiments, we use the GTX 590 GPU integrating two GPUs in the same card where each GPU integrates 16 SMs. Each SM (see Figure. 3.9) is composed of 32 cores, four Special Function Units (SFUs), 16 Double Precision (DP) Units, 16 LD/ST Units, four texture units, two warp schedulers and two dispatch units. Texture units are dedicated hardware units which improve the data access that present 1D, 2D or 3D spatial locality. The warp schedulers and dispatch units are responsible to schedule the threads execution on cores, where each 32 threads are grouped into a warp. Fermi GPU may issues two warps to execute concurrently per clock cycle. Moreover, one SM includes 32 KB of registers, a 64 KB of configurable shared memory/L1 cache and a uniform cache. The configurable shared memory/L1 cache can be configured as 48 KB/16 KB or 16 KB/48 KB. This feature gives more flexibility to the developer toward more data control via shared memory or less control with less programming effort via L1 cache. Each GTX 590 GPU includes also 768 KB of L2 cache and 1.5 GB of global memory. Fermi GPUs integrates also Direct Memory Access (DMA) engines. The DMA engines may be considered as other sources of parallelism since they allow to overlapping computation and asynchronous memory transfers. In mid-range Fermi GPU such as GTX family, only one DMA is present while in high-end Fermi GPU such as Tesla family, we find two DMA engines. Besides its power to execute massive data-parallel applications, Fermi GPU supports also task-parallel applications by means of concurrent streams – command queues that enqueue kernel execution commands and memory transfers commands. Fermi GPU may support up to 16 concurrent streams but they are multiplexed into a single hardware queue which introduces false dependencies between streams execution.

With regard to the runtime, Fermi GPU supports at most 8 active thread blocks for a total of 1536 threads per MP. This runtime constraint has to be considered by the programmer when creating thread blocks and assigning workload to threads.

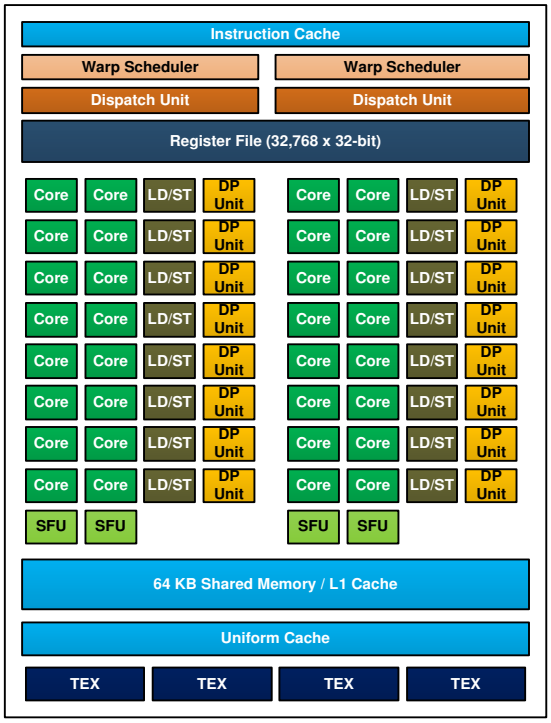


Figure 3.9 Fermi SM Architecture

NVIDIA Kepler GPU

The NVIDIA Kepler GPU codenamed GK110 (NVIDIA, 2012) implements additional hardware resources and provides more parallelization abilities compared to Fermi GPU. The Kepler GPU is composed of a number of MPs with a different structure named SMX. For our experiments, we use the GTX 780 GPU with 12 SMXs. Each SMX (see in Figure. 3.10) is composed of 192 cores (6x more cores than Fermi), 16 SFUs (4x more than Fermi), 64 DP Units (4x more than Fermi), 32 LD/ST Units (2x more than Fermi), 16 texture units (4x more than Fermi), four warp schedulers and eight dispatch units. According to the number of warp schedulers, Fermi GPU may issues four warps to execute concurrently per clock cycle. Moreover, one SMX includes 64 KB of registers (2x more than Fermi), a 64 KB of configurable shared memory/L1 cache and a 48 KB constant cache. The configurable shared memory/L1 cache can be configured as 48 KB/16 KB, or 32 KB/32 Kb or 16 KB /48 KB. The GTX 780 GPU includes also 1536 KB of L2 (2x more than Fermi) cache and 3 GB of global memory. Like Fermi GPUs, Kepler GPUs integrate also one or two DMA depending on the GPU range. Like Fermi, Kepler GPU supports also task-parallel applications by allowing 32 concurrent streams which are managed by means of HyperQ – a technology that implements

32 different hardware queues between host CPU and GPU. This technology enables multiple CPU cores to launch simultaneously kernels on a GPU. This yields to a real full concurrency unlike Fermi GPU. Moreover, Kepler GPU introduces the dynamic parallelism for the first time in GPU where one kernel may launch new kernels on the fly from the GPU without the need of the host CPU.

With regard to the runtime, Kepler GPU supports more active threads than Fermi as 16 active thread blocks of a total of 2048 threads per SMX. This runtime constraint gives more flexibility and better load balancing compared to Fermi architecture.

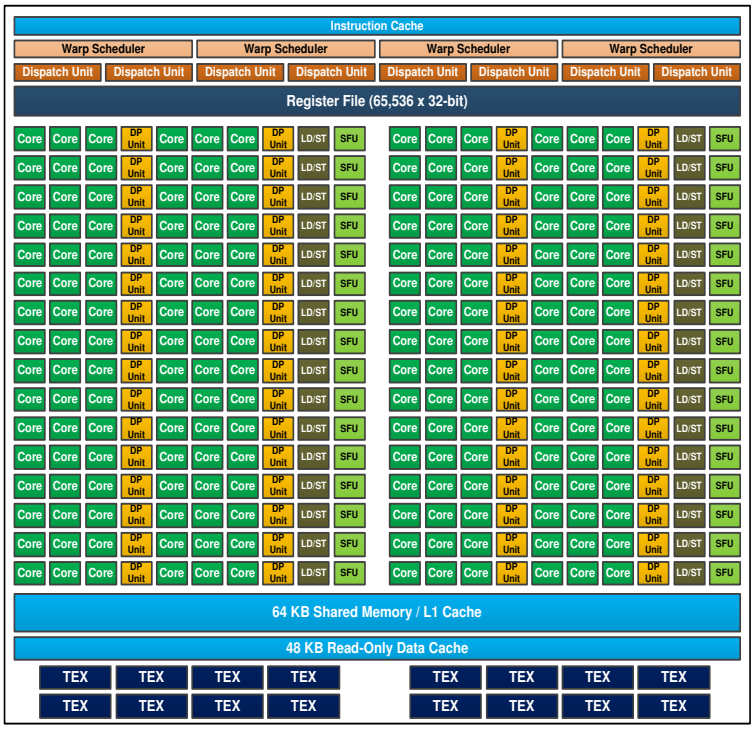


Figure 3.10 Kepler SMX Architecture

NVIDIA Maxwell GPU

The NVIDIA Maxwell GPU codenamed GM107 (NVIDIA, 2014) is the latest architecture released by NVIDIA. For our experiments, we use the GTX 750 GPU which is intended to be used in power-limited devices such as notebooks. The main benefit of Maxwell architecture is its ability to guaranty a high performance rate per watt. Maxwell is able to deliver 2 times the performance per watt compared to Kepler (NVIDIA, 2014). In Maxwell, the MP is named SMM and integrates 128 cores divided into four separate processing blocks of 32 cores each

(see Figure. 3.11). Each processing block integrates its own instruction buffer, warp scheduler and two dispatch units. Each two processing blocks share four texture units and a texture cache combined with an L1 cache. Unlike Fermi and Kepler, Maxwell integrates a separate unit for the shared memory of a larger size of 64 KB. Moreover, Maxwell integrates a larger L2 cache of size of 2048 KB. Maxwell architecture has the same compute capability as Kepler by supporting the dynamic parallelism and integrating a HyperQ.

With regard to the runtime, Maxwell GPU supports the same number of active threads per SMM as Kepler but 32 of active thread blocks. This runtime constraint gives more flexibility and better load balancing compared to Fermi architecture.

Figure 3.11 Maxwell SMM Architecture

Table 3.1 GPU Architectures Specifications

GPU Architecture	Fermi (GTX 590)	Kepler (GTX 780)	Maxwell (GTX 750)
CUDA Compute Capability	2.0	3.5	5.0
# of MPs	2x 16	12	4
# of Cores	2x 512	2304	512
GPU Clock Rate (MHz)	1225	900	1320
Global Mem. (MB)	2x 1536	3072	2048
L2 Cache (KB)	768	1536	2048
Max Shared Mem. (KB)	48	48	64
L1 Cache (KB)	48	48	24
Texture/R.O Cache (KB)	48	48	24
Register File (KB)	32	64	64
# of DMA Engines	1	1	1

Table 3.2 GPU Runtime Constraints

GPU Architecture	Fermi (GTX 590)	Kepler (GTX 780)	Maxwell (GTX 750)
CUDA Version	6.5	6.5	6.5
Max. # of Thread BLocks	8	16	32
Max. # of Threads	1536	2048	2048
Max. # of Threads/Block	1024	1024	1024

3.5.3 Embedded Parallel Platforms

In this section, we give the architecture details of two manycore embedded platforms that we use to implement the studied applications. These platforms are namely STHOTM platform, a CPU core-based platform designed by STMicroelectronics and Tegra K1, a GPU core-based platform designed by NVIDIA.

STHORM Platform

The STHORM platform is a low power many-core CPU platform. The different modules of this platform are represented in Figure. 3.12. The STHORM architecture (Melpignano et al., 2012) features an ARM host CPU running at 667 MHz, a global memory and a four cluster fabric subsystem. Each cluster features one cluster controller and 16 processing elements (PEs) sharing a 256 KB local scratchpad memory. The cluster controller consists of a controller processor (CP) and two Direct Memory Access (DMA) modules responsible of asynchronous data transfers between the global and the local memory. All processors involved in the fabric subsystem are STxP70 32-bit RISC processors running at 450 MHz.

The STHORM platform shares some features with the GPU architecture as the processors organization and the scratchpad memory. However, it presents different features as the PEs are mono-thread versus multithreaded PEs in GPU. Moreover, STHORM follows the MIMD execution model versus a SIMD execution model followed by GPU. This last feature makes STHORM more suitable for data-oriented applications with complex control flow than GPU.

As programming model, customized OpenCL APIs are provided to program parallel applications on STHORM platform.

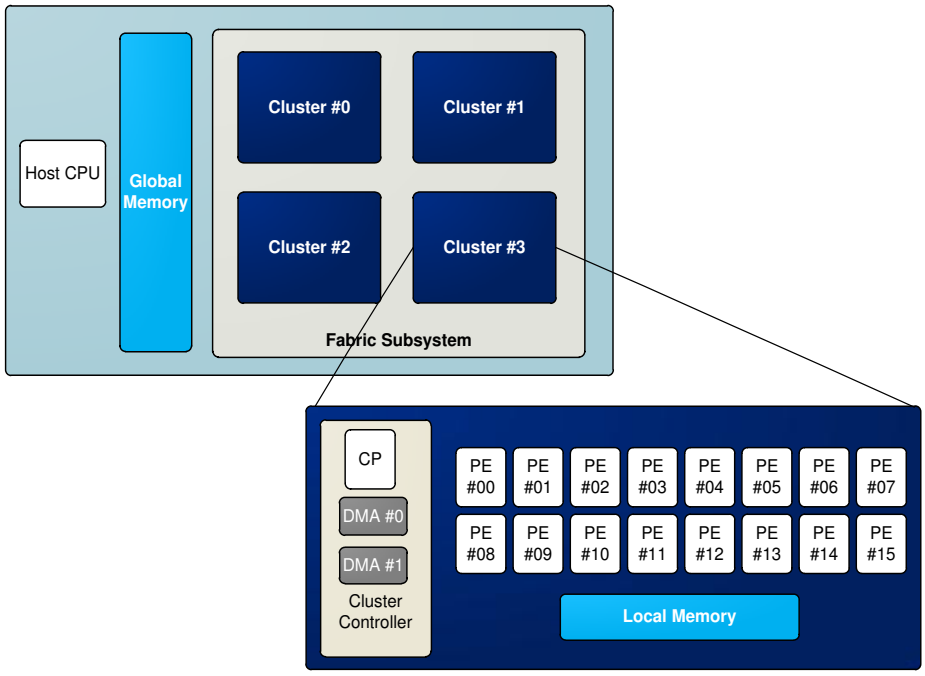


Figure 3.12 STHORM Architecture

Tegra K1 Platform

It is a low power manycore GPU embedded platform designed for mobile applications. Tegra K1 is based on the Kepler architecture and is the first embedded platform supporting CUDA (NVIDIA, 2013). It features one SMX with 192 CUDA cores and a quad core ARM Cortex A15 CPU (see Figure 3.13). Tegra K1 has the same compute capbility as Kepler GPU but working at low power. All the architecture specifications are summarized in Table 3.3.

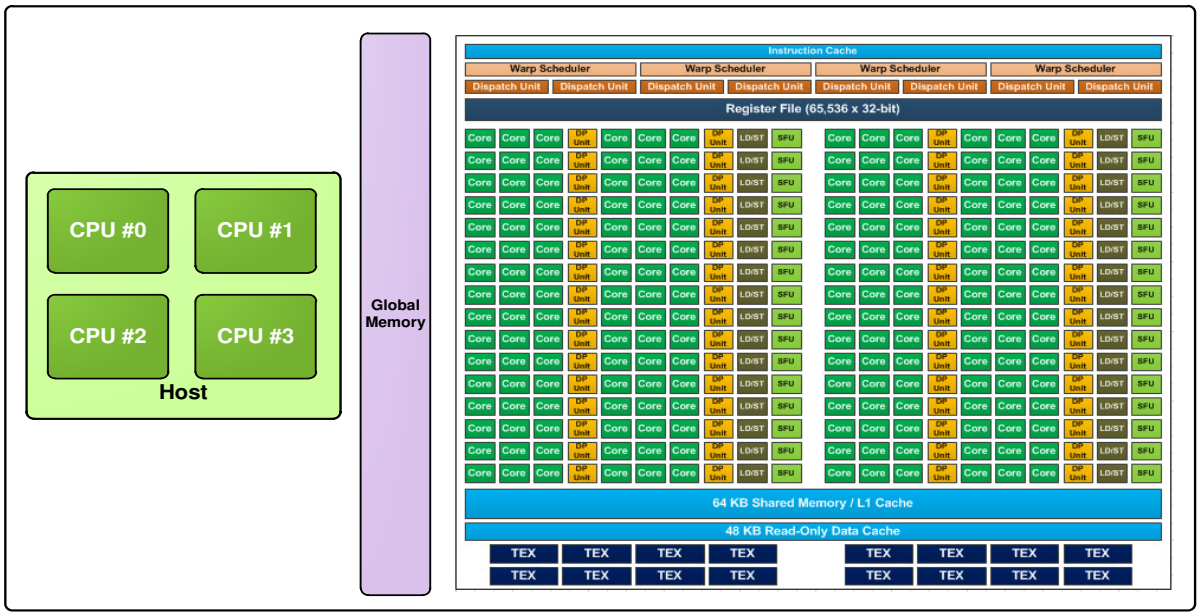


Figure 3.13 Tegra K1 Architecture

Table 3.3 Tegra K1 Architecture Specifications

# of CPUs	4
CPU Clock Rate	2.3 GHz
Memory Size	1700 MB
CUDA Version	6.0
Compute Capability	3.2
Global Mem.	1746 MB
# of SMs	1
# of CUDA Cores	192
GPU Clock Rate	852 MHz
L2 Cache Size	128 KB
Shared Mem. Size	48 KB

3.6 Parallelization Approach

In this section, we describe our approach to parallelize an application for a target parallel platform. This approach may be divided into two steps :

1. We analyze the application regarding the computation complexity, the involved data structures and dependencies. At the end of this step, we extract all possible parallelization opportunities,
2. We employ appropriate parallelization strategies to map the parallelization opportunities to the target platform. The choice of the convenient parallelization strategies depends on the parallelism models found in the application and the target hardware features : number of cores, memory hierarchy and the computational model.

3.6.1 Algorithm Analysis

First, we study the types of operations and data structures involved in each application. Figure 3.14, Figure 3.15 and Figure 3.16 represent both operations and data structures of respectively CED, Thinning and EDT operations. In these figures, a 2D grid represents an image, a dark blue square represents the current processed pixel, a light blue square represents the additional pixel needed for computation and an arrow represents the execution flow.

CED Algorithm Analysis

We can distinguish in CED four main operations : (1) convolution,(2) magnitude calculation, (3) non maximum suppression and (4) edge pixel connection. In Gaussian Blur and Gradient Calculation stage, two successive 1D convolutions are processed, one on the rows and one on the columns. Each 1D convolution may be executed separately on each pixel but the convolution on columns must be executed after the convolution on rows because of the inter-operation dependencies.
Magnitude Calculation may be processed separately for each pixel and needs only the two-directional gradient values at the current pixel index. The Non Maximum Suppression stage needs the current pixel value from both GradientY and GradientX image and the magnitude of the gradient at the current pixel index and of the eight neighbor pixels.
Edge pixels connection goes through each dark blue pixel and checks if one of its neighbors may be connected to it. The traversal of the image follows an unpredictable path since it depends on the pixel values. Therefore, this operation is maintained sequential.

The other operations may be fully parallelized as well. Still, the main challenge of an efficient parallelization of CED is how to maintain a good data locality and hide any data transfer overhead. As a solution, we implement two optimization techniques that are adapted for each family of target architectures. These techniques are mainly : (1) operations fusion and (2) data transfer and computation overlap.

Thinning Algorithm Analysis

The thinning stage performs an iterative processing via an unbounded loop where the number of iterations depends on the processed data. Nevertheless, Thinning application processing may be performed in each pixel separately which makes the application to run in parallel. Similar to CED, thinning also accesses to a large data structures and this access has to be efficient by increasing data locality. To overcome this challenge, we implement a tiling technique which divides image into slices that are processed in parallel with a high spatial data locality.

EDT Algorithm Analysis

In this application, each edge is processed separately. For each edge, a distance envelope is computed based on a precalculated distance table. The mapping of this table and the distance update of each neighboring pixel may be further processed in parallel. The main issue of the parallelization of such application is the work load unbalancing since the edge sizes are variable. To solve this issue, we use mainly two optimization techniques : data arrangement and efficient scheduling.

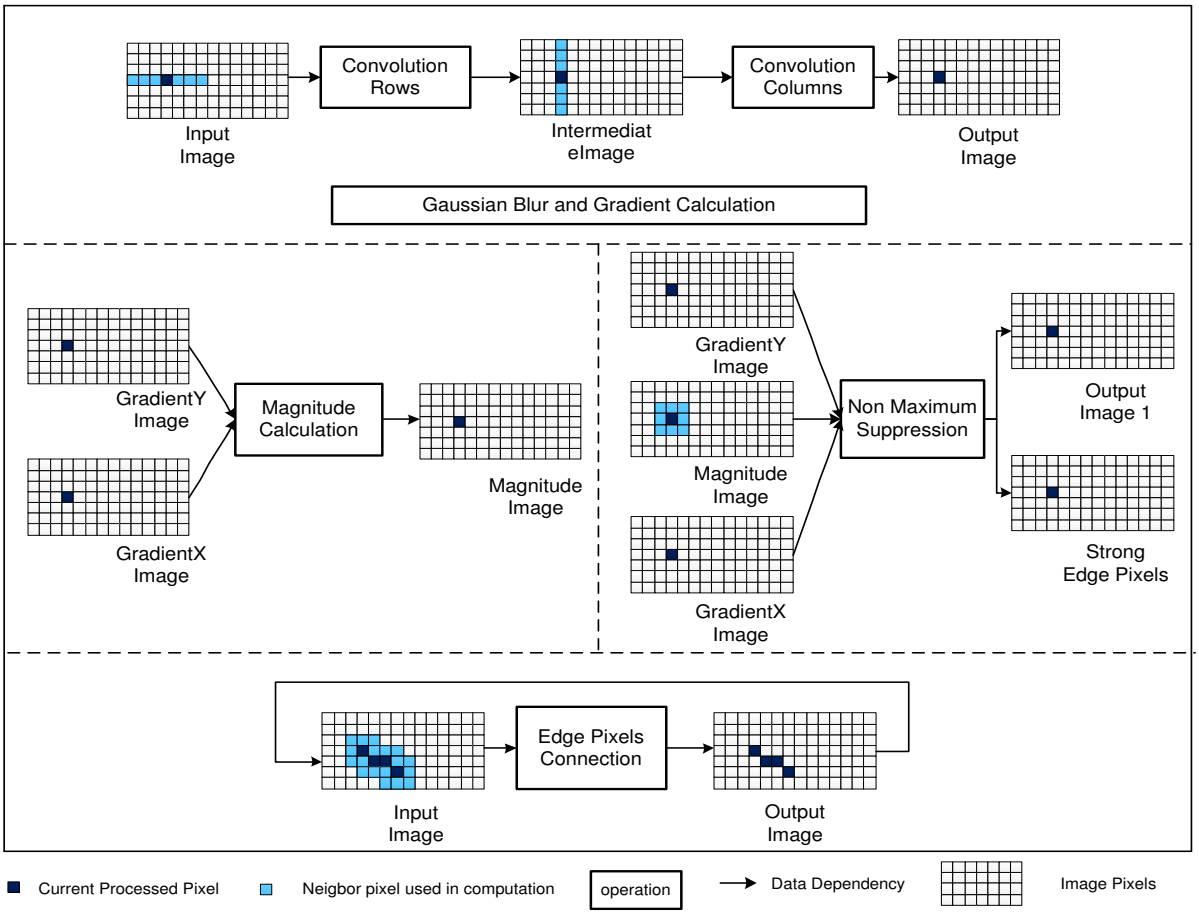


Figure 3.14 CED Algorithm Analysis

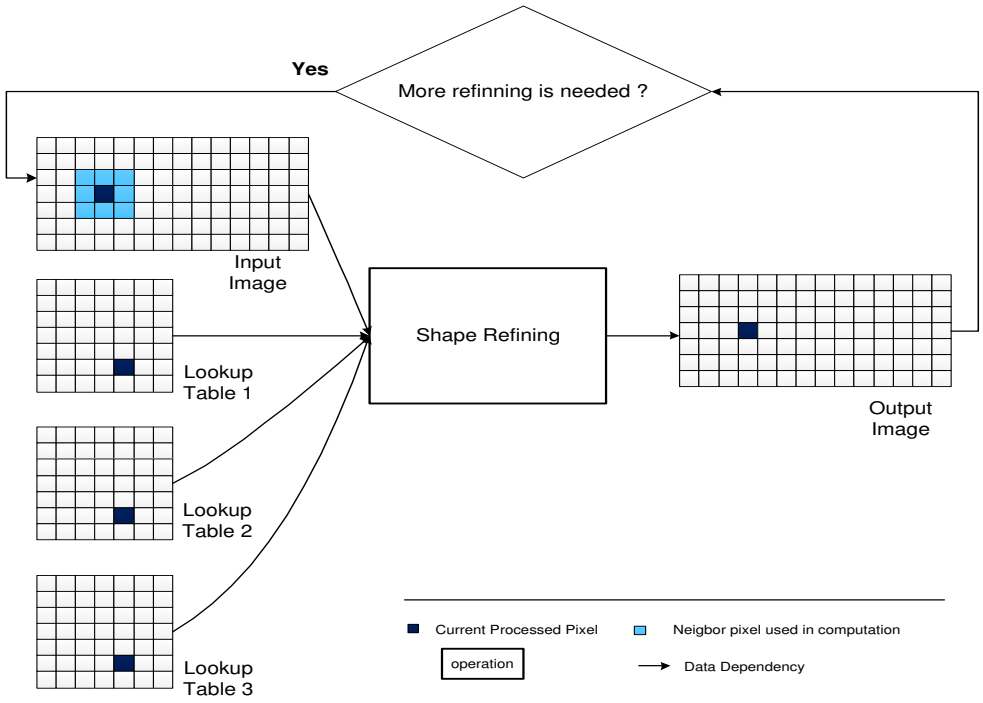


Figure 3.15 Thinning Algorithm Analysis

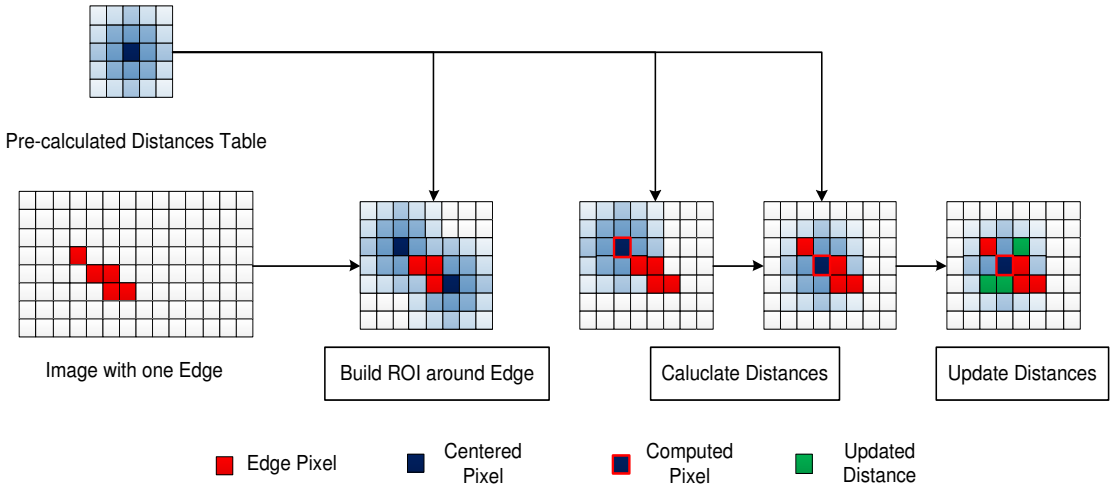


Figure 3.16 EDT Algorithm Analysis

Summary of Computational Features of Studied Algorithms

The studied applications present a number of different computational features summarized in Table 3.4.

- CED combines both data-parallel processing (filtering operations and non maximum suppression) and sequential processing (connecting edge pixels). The processing is mostly regular with the presence of some conditional processing in non maximum suppression stage. The data accesses take the major part of CED execution time so the classification of CED as memory bound application like several image processing applications.
- Shape refining applies iteratively a number of morphological operations. The number of iterations depends on the processed data. This type of processing is known as unbounded loop processing. Like CED, shape refining is also a memory bound application.
- EDT follows a regular processing but suffers from poor load balancing processing due to the difference in size of computed distance region. EDT is based on three main operations : (i) distance computation, (ii) distance comparison which is translated to more control and (iii) distance update which leads to extra data access.

Table 3.4 Computation Features of the Studied Algorithms

Canny Edge detection	**Shape Refining**	**Euclidean Distance Transform**
Filtering Operations	Morphological Operations	Distance Calculation
Regular processing + some control	Iterative with unbounded loop + some control	Regular processing + Control
Data-parallel + sequential Memory bound	Data-parallel Memory bound	Data-parallel + Unbalanced work load Highly memory bound

3.6.2 Parallelization Strategies

The studied target platforms may be grouped into two main categories : (1) simple-level parallel platforms and (2) two-level parallel platforms. The multicore CPU platform is considered as a simple level parallel platform since all launched threads are exposed at the same level. GPU-based platform and manycore cluster-based platform such as STHORM are considered as two-level parallel platforms. The launched threads are organized in a two-level hierarchy. At the first level, the computation is distributed among a number of thread blocks at coarse-grain. Each block has its own workspace defined usually by the shared memory space allocated to it. All threads belonging to this thread block compose the second level of the

hierarchy. Once the computation is divided among thread blocks, a further finer-grain distribution of the computation is performed among threads.

Both CED, thinning and EDT applications are implemented in multicore CPU as simple-level parallelism. For CED and thinning, the input image is divided into large slices where each slice is processed in parallel by one particular thread. For EDT, each group of edges is processed by a particular thread. Each thread passes through all pixels of the edge sequentially and compute the distance of the neighbor pixels.

In a two-level parallel platform, processed images in CED and thinning are divided into slices among thread blocks. First, each slice is loaded to the shared memory space allocated by the tread block. Second, threads belonging to the same thread block work in parallel to process the loaded slice. We apply the same principle to EDT where each group of edges is distributed among thread blocks. Each thread block works on one edge at a time and threads belonging to the same thread block work together to map distances in parallel for one edge pixel at a time and repeat the same processing to all edge pixels.

Besides the parallelization strategies, a number of parallelization optimization techniques are mandatory for an efficient parallelization. In the following, we list the used optimization techniques :

- **Opt1 :** We use an appropriate scheduler to ensure better load balancing among PEs. This optimization is applied for the parallelization of EDT on multicore CPU. As OpenMP is used to implement parallel program on multicore CPU, we add the dynamic schedule directive to the parallel loop pragma. Unlike static scheduling where loop iterations are distributed on the PEs based on their iteration number, dynamic scheduling assigns loops iterations on the fly to idle cores.
- **Opt2 :** We add a preprocessing phase for EDT to arrange the data for better work load distribution on the MPs of manycore platforms.
- **Opt3 :** We apply the fusion technique for the parallel implementation of CED on multicore CPU. This technique ensures improved temporal locality.
- **Opt4 :** We apply the tiling technique in the parallel implementation of thinning application on multicore CPU. This technique ensures improved spatial data locality.
- **Opt5 :** We overlap data transfer and computation on manycore platforms by using the DMA mechanism. For NVIDIA GPU-based platforms implementation, the call of DMA is ensured via the use of concurrent streams and asynchronous data copy APIs.

This optimization decreases the data transfer overhead from host to device memory.

- **Opt6 :** We take advantage of the 48 KB and the 24 KB of read only cache (RO Cache) implemented respectively on Kepler and Maxwell GPUs to improve data read from global memory. The use of the RO Cache is simplified in GPU of compute capability 3.5 and higher via a dedicated API. All the implementations preformed on Kepler and Maxwell are using RO Cache.

3.7 Experimental Results

This section presents the results for several implementations of the studied algorithms on the presented parallel platforms, using the parallelization strategies presented in the previous section.

3.7.1 Parallelization Performance of CED

In this section, we provide the experimental results of several parallel implementations of CED on the studied target platforms.

CED on Multicore CPU Platforms

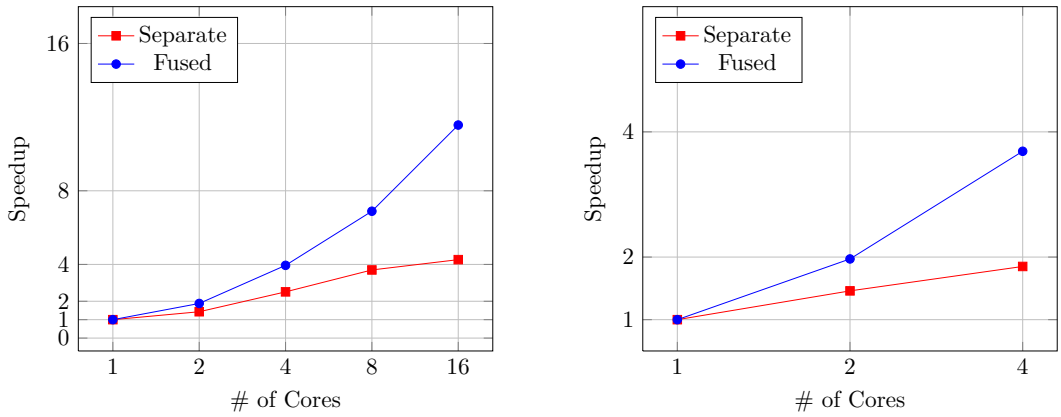
We target multicore CPUs for both HPC and embedded platforms : (1) AMD Opteron 6128 CPU as a HPC platform and (2) the Tegra K1 CPU and the ARM Cortex A15 as embedded platforms.
We implement two parallel versions : (i) a basic implementation where each CED operation is parallelized separately and (ii) an optimized implementation by applying *Opt3* which consists in fusing all the operations into one big operation. This optimization improves the temporal data locality. The execution time and speedup of CED running on AMD Opteron 6128 are illustrated in Table 3.5 and Figure 3.17(a). We show that by applying *Opt3* to the parallelized CED on 16-core AMD Opteron, we increase the speedup from 4.2x in basic implementation to 11.5x in optimized implementation.

Table 3.5 Execution Time in (ms) of CED on HPC Multicore CPU (Image Size = 1024x1024)

# of Cores	**1**	**2**	**4**	**8**	**16**
Separate	448	313	178	121	105
Fused	324	172	82	47	28

Table 3.6 Execution Time in (ms) of CED on Tegra K1 CPU (Image Size = 1024x1024)

# of Cores	1	2	4
Separate	429	292	231
Fused	207	105	56



(a) Speedup of CED on AMD Opteron 6128

(b) Speedup of CED on Tegra CPU

Figure 3.17 Speedup of CED on Multicore CPU Platforms

CED on Manycore Platforms

We implement a parallel version of CED on manycore GPU. The main performance limitation of a parallel implementation of CED on GPU is the runtime overhead introduced by data transfer between host and the GPU. Both computation and data transfer runtime distribution of CED running on GTX 780 are illustrated in Figure 3.21(a). Host2Device and Device2Host denote respectively for data transfer from host to GPU and data transfer from GPU to Host. The data transfer takes around 50% of the total CED runtime and this percentage increases as the image size increases. Since CED reads one input image and generates two images (one represents the candidate pixels to be part of the edge set as white pixels, and the other represents only strong edge pixels as white pixels), the runtime of Device2Host is almost 2x the Host2Device. In order to reduce this overhead, we apply *Opt5* to overlap computation and data transfer by launching concurrent streams. Figure 3.18, Figure 3.19 and Figure 3.20 represent the timeline of respectively basic implementation without overlap, optimized implementation with computation and data read overlap, and optimized implementation with computation and data write overlap where each line represents a separate stream and the x axis represents the time. We cannot overlap computation, data read and data write in the studied GPU platforms since they implement only one DMA. However, this is applied in

the implementation of CED on STHORM which integrates two bidirectional DMA engines. Table 3.7 and Figure 3.21(b) represent respectively the Execution time and the speedup of CED on GTX 780 for both the overlapped read and the overlapped write according to the number of concurrent streams. By overlapping the data write and the computation, we reach a peak speedup of 1.3x with 16 concurrent streams compared to a non overlapped version.

Figure 3.26(a) represents the execution time of the parallel implementation of CED on different GPUs for different image sizes. The experiments are applied to an overlapped computation and data write version with 16 concurrent streams. Table 3.8 summarizes both the execution time and the speedup of CED on both HPC multicore CPU and GPU platforms for an image with size of 1024x1024. A peak speedup of 136x versus a sequential version running on one core CPU is reached with GTX 780. This performance shows that GPU is the most efficient platform to accelerate CED.

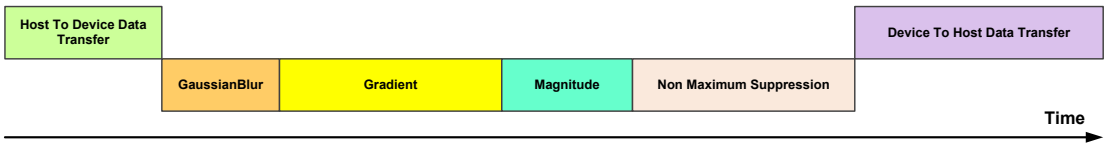


Figure 3.18 CED Without Compute and Data Transfer Overlap

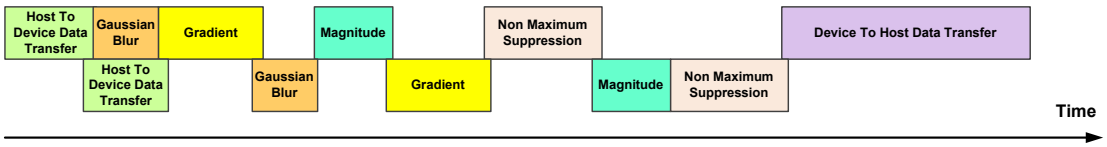


Figure 3.19 CED With Compute and Data Read Overlap

Table 3.7 Execution Time in (ms) of CED on GTX 780 (Image Size = 8192x8192)

# of Streams	**1**	**2**	**4**	**8**	**16**
Overlapped Read	126	117	111	108	106
Overlapped Write	126	116	113	110	98

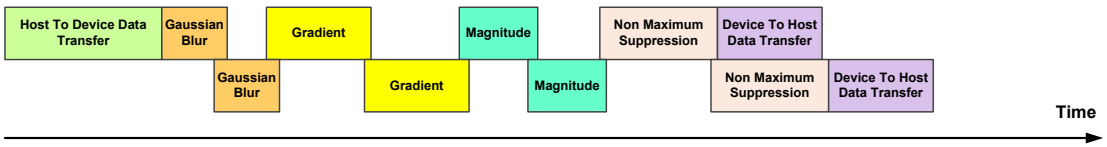
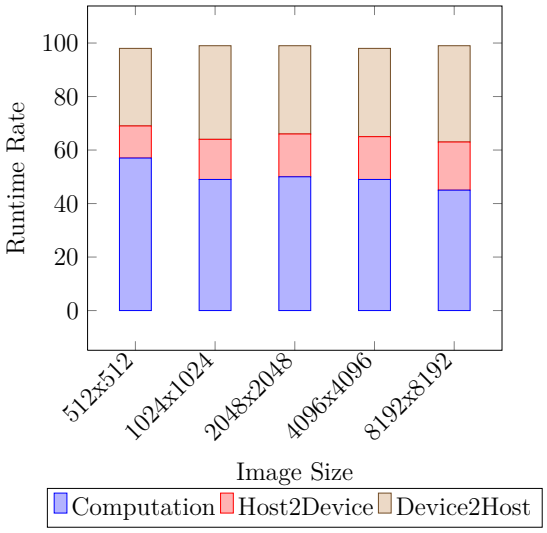
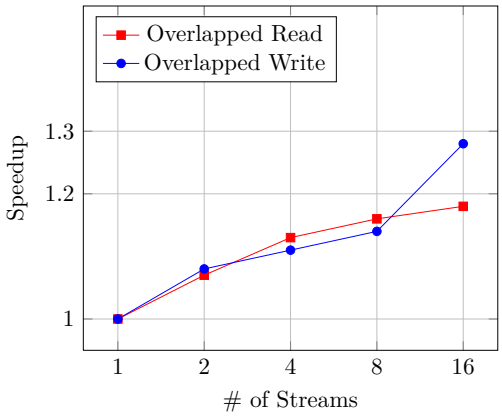


Figure 3.20 CED With Compute and Data Write Overlap



(a) CED : Computation and Data Transfer Runtime Distribution on GTX 780



(b) Speedup of CED on GTX 780 (Image Size = 8192x8192)

Figure 3.21 Data Transfer Analysis and Transfer Improvements on GTX 780

Table 3.8 Parallelization Performance of CED on HPC Platforms (Image Size = 1024x1024)

Platform	**Opteron 6128 1 core**	**Opteron 6128 16 cores**	**GTX 590**	**GTX 780**	**GTX 750**
Execution Time (ms)	324	28	4.35	3.93	4.43
Speedup (x)	1	11.57	106	136	101

3.7.2 Parallelization Performance of Thinning

In this section, we provide the experimental results of the parallel implementations of Thinning application on the studied target platforms.

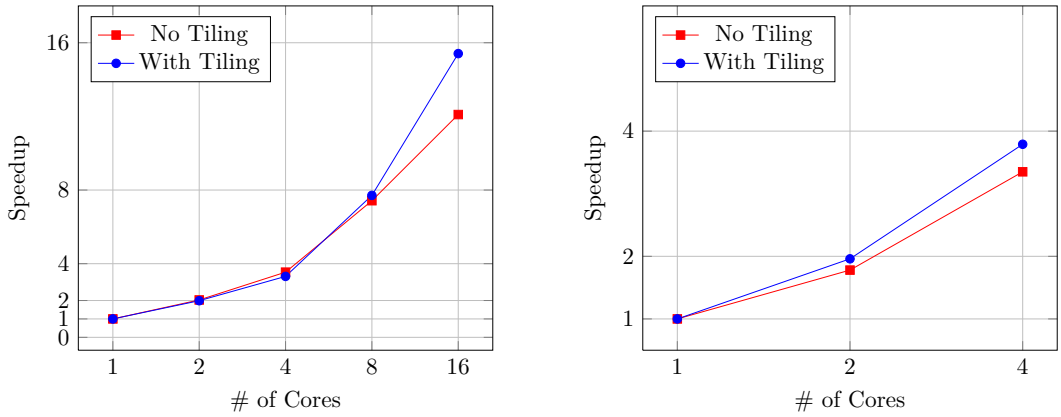
We implement two parallel versions of Thinning application on multicore CPU (basic implementation without tiling and an optimized implementation with tiling (*Opt4*). The execution time of the two implementations are provided in Table 3.9 and Table 3.10 for respectively AMD Opteron 6128 and Tegra ARM Cortex A15. The speedup reached in the two multicore CPU platforms is given in Figure 3.22(a) and Figure 3.22(b). Both implementations are well suited to be parallelized on multicore CPU with a good scalability with respect to the number of cores. The implementation with tiling outperforms the basic implementation when the number of cores exceeds 8 cores (the number of cores per single processor). Since the tiled implementation improves the spatial data locality, the remote data access overhead to a different processor memory is reduced significantly. Hence, tiling optimization is considered as a suitable candidate to accelerate such application on a NUMA multicore CPU.

Table 3.9 Execution Time in (ms) of Thinning on HPC Multicore CPU (Image Size = 1024x1024)

# of Cores	**1**	**2**	**4**	**8**	**16**
No Tiling	557	274	157	75	46
With Tiling	355	178	107	46	23

Table 3.10 Execution Time in (ms) of Thinning on Tegra K1 CPU (Image Size = 1024x1024)

# of Cores	**1**	**2**	**4**
No Tiling	346	194	103
With Tiling	258	131	68



(a) Speedup of Thinning on AMD Opteron 6128

(b) Speedup of Thinning on Tegra CPU

Figure 3.22 Speedup of Thinning on Multicore CPU Platforms

3.7.3 Parallelization Performance of EDT

In this section, we provide the experimental results of the parallel implementations of EDT on the target platforms.

EDT on Multicore CPU Platforms

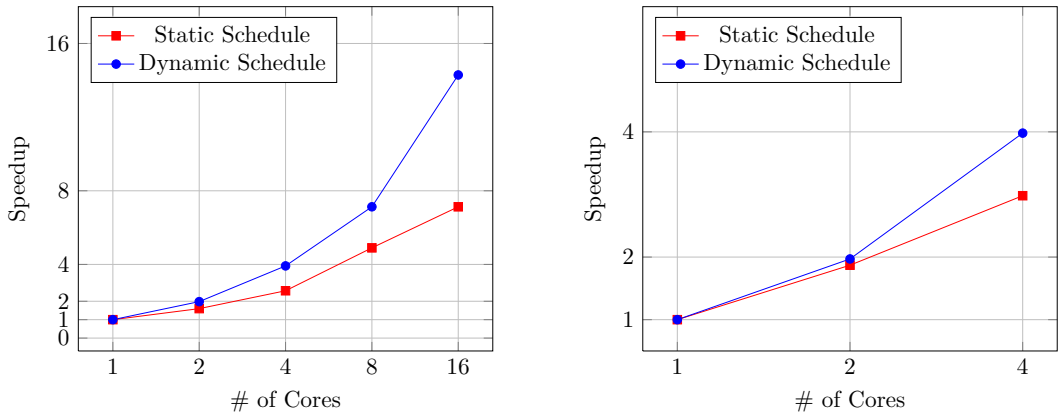
We implement two parallel versions of EDT on multicore CPU. We compare the parallelization efficiency of a basic implementation and an optimized one applying *Opt1*. The execution time of EDT processing 177 edge is given for each implementation in Table 3.11 and Table 3.12 respectively for AMD CPU and Tegra K1 CPU. The different speedups are illustrated respectively in Figure 3.23(a) and Figure 3.23(b). We show that by applying *Opt1* to EDT parallelized on 16-core AMD Opteron, we double the speedup from 7x with static schedule to 14x with dynamic schedule.

Table 3.11 Execution Time in (ms) of EDT on HPC Multicore CPU (# of Edges = 177)

# of Cores	**1**	**2**	**4**	**8**	**16**
Static Schedule	157	98	61	32	22
Dynamic Schedule	157	79	40	22	11

Table 3.12 Execution Time in (ms) of EDT on Tegra K1 CPU (# of Edges = 177)

# of Cores	1	2	4
Static Schedule	227	121	76
Dynamic Schedule	227	115	58



(a) Speedup of EDT on AMD Opteron 6128

(b) Speedup of EDT on Tegra CPU

Figure 3.23 Speedup of EDT on Multicore CPU Platforms

EDT on Manycore Platforms

We target manycore GPUs to implement the basic EDT and optimized EDT applying *Opt2*. Figure 3.24 and Figure 3.25 show the work load distribution on GTX 750 SMs for respectively without *Opt2* and with *Opt2*. The performance comparison of the two implementations on different GPUs with different number of SMs is represented in Figure 3.26(b). The significance of the performance gap between the two implementations is proportional to the number of SMs. An inappropriate work load distribution may harm significantly the performance when SMs are not fully utilized. Hence, we conclude that the work load mapping is a relevant factor to improve perfromances on GPU. However, this performance is still poor compared to the computation capability of the GPU. The maximum speedup shown in Table 3.13 for a parallel version on GTX 780 is about 20x compared to 14x of 16-core multicore CPU speedup. This relative poor performance on GPU is explained by the excessive data exchange between the local shared memory and the global memory to update calculated distances. For such applications which present a high rate of memory access compared to computation intensity, the multicore CPU is more appropriate as acceleration platform.

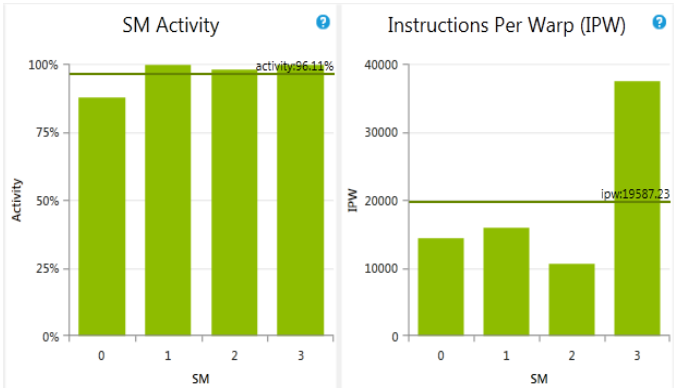


Figure 3.24 EDT Load Balancing Without Data Arrangement on GTX 750

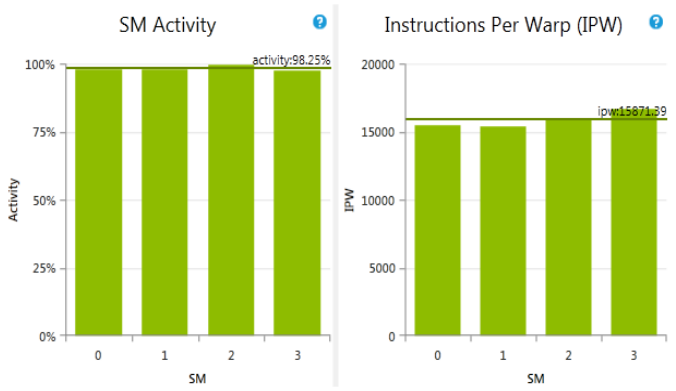
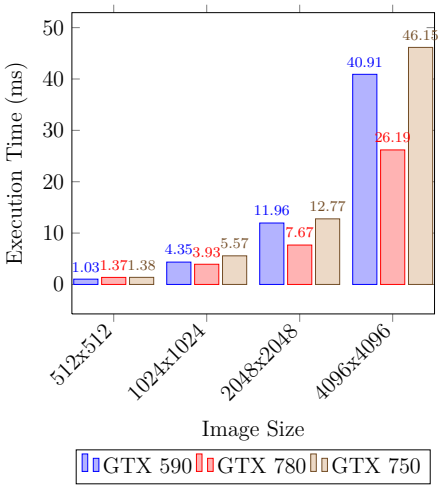
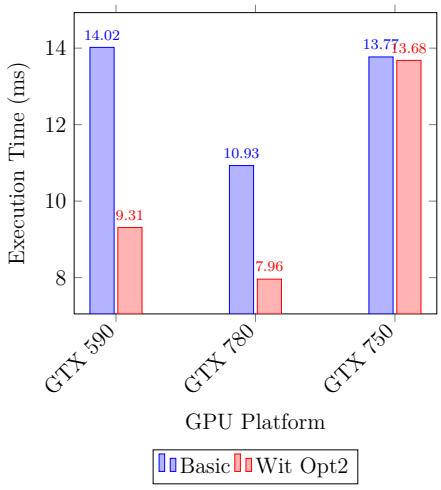


Figure 3.25 EDT Load Balancing With Data Arrangement on GTX 750



(a) Execution Time of CED on HPC GPU Platforms



(b) Execution Time of EDT on HPC GPU Platforms

Figure 3.26 Execution Time of CED and EDT on HPC GPU Platforms

Table 3.13 Parallelization Performance of EDT on HPC Platforms (# of Edges = 177)

Platform	**Opteron 6128 1 core**	**Opteron 6128 16 cores**	**GTX 590**	**GTX 780**	**GTX 750**
Execution Time (ms)	157	11	9.31	7.95	13.68
Speedup (x)	1	14.27	16.86	19.74	11.47

3.7.4 Performance Comparison Between STHORM and Tegra K1

In this section, we provide a performance comparison of two embedded manycore platforms. STHORM as a manycore CPU core-based platform and Tegra K1 as a manycore GPU-based platform. The parallel implementation of two best versions of both CED with thinning application and EDT are evaluated on the two platforms. The execution time and speedup are reported in Table 3.14. The CED experiments are performed in an image size of 512x512. The EDT experiments are performed on 177 detected edges.

Table 3.14 Parallelization Performance of CED+Thinning and EDT on Manycore Embedded Platforms)

Platform	**Tegra 1 core**	**STHORM Sequential**	**Tegra 4 cores**	**Tegra GPU**	**STHORM Manycore**
Execution Time of CED+Thinning (ms)	235.6	2989	62	42	141
Speedup (x)	1	1	3.81	5.6	21.2
Execution Time of EDT (ms)	227	443	58	167	81.83
Speedup (x)	1	1	3.91	1.35	5.41

3.8 Conclusion

In this chapter, we studied the suitability of several parallel platforms to accelerate three fundamental applications used in image processing and computer vision. The studied applications are Canny edge detection, thinning and Euclidean distance transform. Those applications are components of an augmented reality system application used in medical imaging. In order to meet the target application requirements mainly accuracy and speedup, we implemented several parallel versions of the studied applications on various platforms. A number of optimization techniques are applied toward improved performances. These optimizations depends on the application characteristics such as data access pattern and computations, and the compute capability of the target platform such as computation model and memory organization. The experimental results on a wide range of target platforms are provided and can be considered as guidelines to select the appropriate parallel platform to speedup a given computation feature.

The main conclusions that we can extract from this deep study are :

- Overlapping data transfer and computation in GPU represents another source of parallelism which allows to hide extra overhead introduce by data transfer between host and device.
- Inappropriate work load distribution on GPU may harm significantly the performance when MPs are not fully utilized.
- For applications which present a high rate of memory access compared to computation intensity, the multicore CPU is more appropriate as acceleration platform compared to GPU platforms.

CHAPTER 4 PERFORMANCE EVALUATION OF PARALLELIZATION STRATEGIES - CASE STUDY : extCED

4.1 Introduction

Since applications are increasing in complexity and present a wide variety of parallel features, it becomes difficult to decide which parallelization strategy is suitable for a given platform to reach peak performance. The major contribution presented in this chapter is a comprehensive quantitative analysis of a large variety of parallelization strategies and implementing different parallel structures on two most common parallel platforms : a multicore CPU and a manycore GPU using wide spectrum of parallel programming models. We also aim at determining guidelines for the efficient parallelization strategies (parallelization granularity and parallelism model) of a class of image processing applications.

In this context, we developed a substantial number of parallel extended Canny Edge Detection (extCED) implementations on a multicore CPU and on a manycore GPU. Hence, we provide one of largest variety of parallel versions of CED that can exist. Furthermore, we provide a high accurate and fast CED implementation suitable for laproscopic images.

The rest of the the chapter is organized as follows : Section 4.2 presents a detailed state of the art of several performance evaluations and comparisons of parallelization strategies. Section 4.3 presents the case study and the possible parallelization opportunities states previous implementations of CED and describes the CED algorithm ; Section 4.4 exposes the studied parallelization strategies ; Section 4.5 presents the target parallel platforms and the their respective programming models. Section 4.6 presents the design challenges and describes our approach to address the parallelization of as sequential program. Section 4.7 shows the performance evaluation of the studied parallelization strategies. Finally, Section 4.8 draws a number of concluding remarks.

4.2 Related Work

There have been large body of work that compares different parallelization strategies. We can classify them according to three categories (1) parallelization granularity, (2) parallelism model and (3) programming model.

4.2.1 Related Work on Comparing Parallelization Granularities

Several researchers focused on the parallelization granularity, in particular fine-grain through inner loop-level parallelism and coarse-grain through domain decomposition or tiling. The work in (Pratas et al., 2009) compares the efficiency of multicore CPUs vs. Cell/BE vs. GPUs to exploit fine-grain parallelism in a Bioinformatics application. On the other hand, domain decomposition is essentially used for scientific and engineering computations (Guo et al., 2008) and for computer vision applications (Chen et al., 2007b). In particular, image processing algorithms often uses Iterative Stencil Loops (ISL) [1] (Meng and Skadron, 2011) which encourages the use of tiling technique. Experiments provided in (Li and Song, 2004) show that the tiling technique offers a speedup of 1.58 and a maximum speedup of 5.06 over original implementations of sixteen test programs. Several work employ halo [2] for tiling ISLs in order to localize the computations specially for manycore GPUs (Che et al., 2008; Meng and Skadron, 2011).

4.2.2 Related Work on Comparing Parallelism Models

In this subsection, we present some related work dealing with performance evaluation of a number of parallelization strategies covering parallelism models mainly data-parallelism and nested task- and data-parallelism.

Mixing different levels of parallelism is limited in a number of parallel programming model standards. The work presented in (Rodríguez-Rosa et al., 2003) investigates and measures the advantages of mixing task and data parallelism. The experiments are performed with two programming models OpenMP 2.0 and llc as an extension of OpenMP to support both multi-level parallelism and distributed memory architectures. The results demonstrate that llc provides good performance by mixing different levels of parallelism, while OpenMP 2.0 does not benefit from that mixing due to the lack of suited compilers. A more recent work presented in (Dimakopoulos et al., 2008) assesses in details the overhead incurred by nested parallelism implemented with OpenMP 2.5. Several OpenMP compilers and runtime systems were inspected to study how efficiently OpenMP implementations support nested parallelism. The experiments show that most implementations suffer from scalability problems in the case where nested parallelism is enabled and the number of threads exceeds the available processors. In (Tian et al., 2005), design and implementation of a new framework for OpenMP

1. Iterative Stencil Loop (ISL) : loops that iteratively modify the same array elements as a function of neighbouring elements during consecutive processing steps

2. Halo : a region that surrounds each tile which contains a copy of elements of the neighbor tiles needed for computation

2.5 which exploits nested task- and data- parallelism was proposed. The main idea of this existing work is to convert the *parallel sections* construct, used to express task-parallelism, to *parallel for* construct with the capacity to specify the scheduling algorithm in order to achieve a better load balancing. The proposed framework delivers a performance gain of about 40% by exploiting nested parallelism compared with a single level of parallelization. All of the above mentioned work were built around OpenMP and multicore CPUs, while in our case we examine how efficient different parallel programming models and different parallel platforms exploit both data-parallelism and nested task- and data-parallelism.

4.2.3 Related Work on Comparing Programming Models

A comparison between OpenMP 3.0, Cilk (Frigo et al., 1998), Cilk++ (Leiserson, 2009) and Intel Thread Building Blocks (TBB) with respect to scalability regarding task parallelism and in presence of load balancing constraints is reported in (Olivier and Prins, 2010). The Unbalanced Tree Search (UTS) (Olivier et al., 2007) is used as benchmark. The experiments results provided by the paper show that Cilk++ offers the best scalability followed by Cilk then TBB and as last OpenMP 3.0.

In (Kegel et al., 2011b), a comprehensive comparison of Pthreads (Butenhof, 1997), OpenMP, and TBB is made according to : runtime performance, level of abstraction, programming style and programming effort. Different parallelization strategies of a medical imaging algorithm, Positron Emission Tomography (PET), are evaluated for each parallel programming model. The parallelization strategies studied include : loop parallelization, synchronization, and higher level parallelization based on reduction pattern. The comparison results show that the three programming models provide almost similar speedup for the different parallelization strategies except for parallel reduction algorithm where TBB is outperformed by Pthreads and OpenMP on eight cores. Regarding the abstraction level, both OpenMP and TBB describe the parallelism at a higher level compared to Pthreads. While TBB is shown to be more tailored for object-oriented programming style, however, the authors show the need of an additional programming effort compared to OpenMP, but it remains much lower then Pthreads.

Parallelization overhead associated with thread creation, synchronization, threading granularity and scheduling are evaluated for both OpenMP 2.0 and TBB 2.0 in (Marowka, 2010). The experiments are performed with two benchmarks suites OpenMP EPCC micro benchmarks (Bull and O'Neill, 2001) and TBBBench (Marowka, 2010). Moreover, the influence of compilers on measured results is evaluated. The studied compilers are Intel C++ compiler 11.0 and Microsoft Visual Studio C++ 2008. The main conclusions of this work are : (1) OpenMP ove-

rhead is strongly dependent on the compiler, while TBB performances remains independent form compilers, and (2) TBB implements a scheduler that manages task-parallelism model in more efficient way than OpenMP.

While the above mentioned work provides a preliminary evaluation of a number of parallelization strategies and their implementation by different programming models, they are still limited to multicore CPUs. In contrast, our proposed work comes to compare a wide range of parallelization strategies on both multicore CPUs and manycore GPUs. Our main goal is to allow the developer to implement the most effective parallelization strategies on current heterogeneous parallel platforms integrating multicore CPUs and manycore GPUs.

In that direction, an in-depth performance comparison between CUDA and OpenCL on NVIDIA GPU is provided in (Karimi et al., 2010). The measured performances involve data transfer times to and from the GPU, kernel execution times, and the whole application execution times. As benchmark, Adiabatic QUantum Algorithm (AQUA) is used. The experiments report that CUDA outperforms OpenCL in both data transfer and kernel execution.
In addition, a number of publications are comparing performances between multicore CPUs and manycore GPUs. In (Che et al., 2008), a CUDA-based implementation on manycore GPU is compared to OpenMP-based implementation on multicore CPU for a variety of general-purpose applications. In this work, the authors examines the programmability and runtime performance of CUDA compared to OpenMP for different data structures and memory access patterns. In (Che et al., 2009), Rodinia, a new benchmark suite for heterogenous computing, is introduced. The suite consists of four applications and five kernels. Two implementations of Rodinia are available : (1) OpenMP-based implementation targeting multicore CPUs and (2) CUDA-based implementation targeting manycore GPUs. The benchmark suite offers a wide range of workloads covering various types of parallelism, data access patterns, and data-sharing characteristics. The goal of this work is to facilitate performance evaluation of different parallel communication patterns, synchronization techniques and power consumption for both multicore CPUs and manycore GPUs.

Despite the interesting comparison of a number of parallelization strategies between multicore CPUs and manycore GPUs, these works are limited only to two parallel programming models OpenMP and CUDA while our work targets OpenMP and TBB for mutli-core CPUs and, CUDA and OpenCL for manycore GPUs.

Through this extensive summary of the related work, we can build some preliminary conclusions around the efficiency of a number of parallelization strategies in a given context (parallel

structure and parallel architecture). However, we have not a comprehensive overview of the common parallelization strategies running under different contexts. Hence, the focus of the proposed work is to fulfill this need.

4.3 Motivation and Example : Extended Canny Edge Detection (extCED)

As the promise of parallelism is attracting researchers and developers for the last decades, the programming of current applications exploits further the parallelism that may be present in those applications. The parallelism opportunities may vary depending on the application and the application field while several parallelism opportunities may be shared by different applications belonging to distinct application fields. One of the biggest limitation to reach high performance execution in parallel computing that it is currently unclear how to express the parallelism opportunities best. To overcome this limitation and based on the fact that applications may share similar parallelism opportunities, we decided in this work to see if there are lessons we can learn for the future of parallel computing.

Our goal is to draw general conclusions about the efficiency of existing parallelization strategies. To do so, we study a particular application, the Extended Canny Edge Detection (extCED), which exposes various parallelism opportunities that are representative of a wide range of parallel features which may be present in a large set of applications. Then, we develop a large variety of parallelization strategies that may express the different parallelism opportunities and we compare their performances according to runtime and scalability in different contexts of execution (multicore CPU and manycore GPU).

4.3.1 Case Study : Extended Canny Edge Detection (extCED)

CED is used as a preprocessing phase in several computer vision applications. As mentioned in the previous chapter, CED has been used as a first step in the process of identifying and tracking instruments during laparoscopic surgery. Since our work targets medical imaging, which requires a high accurate detection, we add another stage responsible of refining the edges' shapes. This stage involves a thinning morphological operation.

We can distinguish mainly three main parts in the extCED algorithm, depending on the way in which parallelism can be exploited (see Figure 4.1). Part 1 is purely parallel, and can be equally distributed over different threads. Part 2 is strictly sequential, and Part 3 consists

of a sequential loop including pure data-parallel sub-parts (see Listing 4.1 [3]). Part 1 can be further divided into two main sub-parts : pure data-parallel sub-parts and nested task- and data-parallel sub-parts. Based on this classification, we experiment several parallelization strategies.

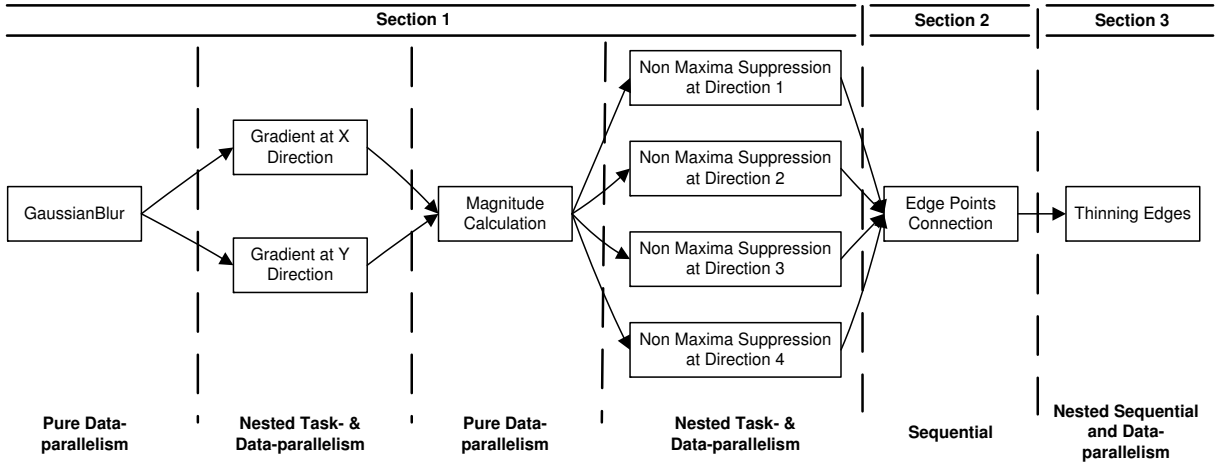


Figure 4.1 Canny Edge Detector Diagram

3. For convenience, we use OpenMP constructs in the listings to describe the parallel structures of the different program parts.

Listing 4.1 Nested Sequential and data-parallelism

```
while(condition){
  //Fist sub-iteration
  #pragma omp parallel for
  for(row=0; row<imHeight; row++){
    for(col=0; col<imWidth; col++){
      out1(row, col) = function_1(in1);
    }
  }

  //Second sub-iteration
  #pragma omp parallel for
  for(row=0; row<imHeight; row++){
    for(col=0; col<imWidth; col++){
      out2(row, col) = function_2(out1);
    }
  }

  condition = function_3(in1, out2);
}
```

4.4 Parallelization Strategies For extCED

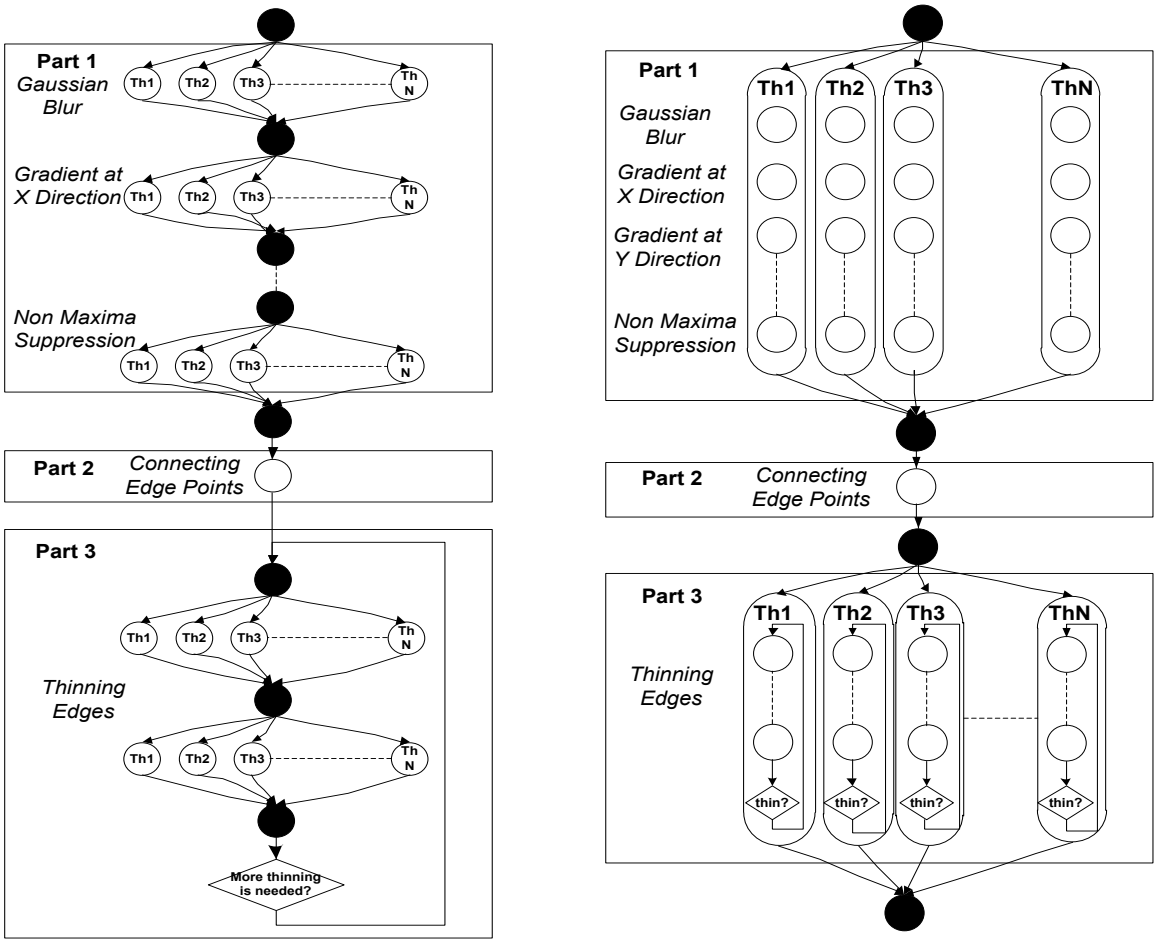
We can classify the studied parallelization strategies according to the parallelization granularity, parallelism models.

4.4.1 Parallelization Granularities

The studied parallelization granularities are : (1) fine-grain and (2) coarse-grain.

- **Fine-grain parallelization :** it consists of parallelizing separately each loop with no dependencies across iterations (see Figure 4.2(a)). The particular feature of fine-grain parallelization strategy is the high reusability since each loop may be a stand alone image processing operation that can be present in more than one image processing algorithm. So parallel version of this operation can be reused more than once in different applications without any modification. However, this strategy may suffer from the overhead due to the thread's fork-join operations, and from poor data locality.
- **Coarse-grain parallelization :** the program is parallelized at outermost level : the

input image is equally divided in sub-images or tiles then the whole program processes separately each of them (see Figure 4.2(b)). The input image of the CED is split into sub images of equal sizes, and a number of image processing operations are packed into one big operation which processes each sub image independently and in parallel. The implementation of the data decomposition strategy needs additional coding effort to split and merge the sub-images. It is also necessary to assess the presence of dependencies between neighbouring sub-images : in this case, the partitioning strategy has to be tuned to minimize the impact of dependencies on performance. However, domain decomposition offers a good runtime performance by avoiding additional overhead caused by fork-join operations and by increasing the data locality.



(a) Fine-grain

(b) Coarse-grain

Figure 4.2 Parallelization Granularities

4.4.2 Parallelism Models

The studied parallelism models are : (1) pure data-parallelism, and (2) nested task- and data-parallelism.

- **Pure data-parallelism :** refers to work units executing the same operations on a set of data. The data are typically organized into a common structure as arrays or vectors. (Part1, Figure 4.1) Each work unit [4] performs the same operations as other work units on different elements of the data structure. The first concern of this type of parallelism is how to distribute data on work units while keeping them independent. The relevant advantage of data parallelism is that it exploits a large number of processing elements of the target hardware platform to offer a maximum parallelism.
- **Nested task- and data-parallelism :** in most of applications, only one parallelism model is not enough to achieve high degree of parallelization. A combination of different parallelism models will be needed under a nested task- and data parallelism model. In particular, CED algorithm shows that nested task- and data-parallelism in part 1 : Gradient Calculation can be processed as two parallel tasks, where each task processes a Gradient Calculation (Part1, Figure 4.1) at one direction (see Listing 4.3). Each task in its turn encapsulates loops that can be parallelized as the data parallelism trend (see Listing 4.2). On the other hand, Non Maxim Suppression (Part1, Figure 4.1) integrates the nested task and data parallelism but with four tasks that can be processed in parallel (see Listing 4.4).

Listing 4.2 Data-parallelism

```
void Gradient_X(vars, numThreads){
  #pragma omp parallel for
  for(row=0; row<imHeight; row++){
    for(col=0; col<imWidth; col++){
      ...
    }
  }
}
```

4. Work unit : the unit of processing, it may be a task or a thread performing in parallel manner.

Listing 4.3 Nested task- and data-parallelism with two tasks

```
#pragma omp parallel sections num_threads(2)
{
  #pragma omp section{
    Gradient_X(vars, numThreads/2);
  }
  #pragma omp section{
    Gradient_Y(vars, numThreads/2);
  }
}
```

Listing 4.4 Nested task- and data-parallelism with four tasks

```
#pragma omp parallel sections num_threads(4)
{
  #pragma omp section{
    NonMaximaSuppression_1(vars, numThreads/4);
  }
  #pragma omp section{
    NonMaximaSuppression_2(vars, numThreads/4);
  }
  #pragma omp section{
    NonMaximaSuppression_3(vars, numThreads/4);
  }
  #pragma omp section{
    NonMaximaSuppression_4(vars, numThreads/4);
  }
}
```

4.5 Parallel Platforms and Programming models

In this work, we target two different mainstream parallel platforms mainly multicore CPU and manycore GPU. Each architecture presents some advantages and limitations. While multicore CPU has the ability to execute efficiently both control- and data-oriented parallel applications but with a limited number of parallel work units, manycore GPU is more suitable for data-oriented parallel applications with large number of work units but presents poor performances for control-oriented applications.

4.5.1 Multicore CPU and Programming Models

Multicore CPU refers to general purpose processor integrating multiple cores in the same die. In general, these cores are identical and they are based on x86 architecture. The multicore CPU is a cache-based architecture which is classified as shared memory system. Since multicore CPU integrates an independent control unit for each core, it is able to handle both task parallelism and data parallelism efficiently. However, the number of cores is limited to order of ten cores which limits the number of parallel work unit.

To program the multicore CPU, we study two programming models namely OpenMP and TBB. Both are high-level programming models but follow different strategies. While OpenMP is a directive based programming model, TBB is a template library-based programming model. As consequence, the performance of OpenMP program depends on the used compiler unlike TBB program where the performance stays unchanged with different compilers.

4.5.2 Manycore GPU and Programming Models

Manycore GPU is a scratchpad memory-based architecture which integrates a hierarchical memory system. Manycore GPU consists of a set of streaming multiprocessors (SM), each composed of a number of streaming processors (SP) or cores. SPs from the same SM share a fast on-chip memory (scratchpad shared memory). All SPs can access the global memory which is much larger than the shared memory with order of hundreds of MB. However, the global memory is an off-chip memory with higher latency. The GPU architecture is essentially composed of hundreds of functional units and only few control units that it makes it more suitable for fine-grain data parallel processing.

To program manycore GPU, we study two programming models namely CUDA and OpenCL. Both are considered as relatively low-level programming models. CUDA can be used in two different ways (1) via the runtime API, which provides a C-like set of routines and extensions, and (2) via the driver API, which provides lower level control over the hardware, but requires more code and programming effort. OpenCL API has a lot of similarities with CUDA driver API which leads to additional effort to write a parallel program on GPU.

4.6 Design Challenges and Approach

The selection of parallelization strategies represents currently a key issue for the achievement of required performances for image processing applications. Most of these applications expose various parallelism opportunities and hence a mixture of parallelization strategies can be

exploited. The developer may face several choices to parallelize this kind of applications. He has to decide which parallelization strategy will be the best suitable. Mostly, this decision is not trivial in particular with the large number of possible strategy combinations and different target hardware platforms. In order to ease the parallel programming task and guide this decision, we adopt a particular approach (see Figure 4.3).

This approach deals with the main parallelization strategies that need to be explored for an efficient parallel implementation of image processing applications. The approach is built in four stages :

- Stage 1 : it consists in analyzing the application according to dependencies either at the execution flow or at data decomposition levels. The analysis at the execution flow level finds the data dependencies that occur during the application execution while the analysis at data-decomposition level defines how the data may be divided for further parallelism.
- Stage 2 : it consists in selecting the parallelization granularity depending on the level of the dependency. In fine-grain the parallelization is performed separately for each individual task. In coarse-grain the parallelization is performed by splitting the input data structures into sub-data structures and grouping a set of tasks into one big task to be applied on each sub-data structures.
- Stage 3 : Once the granularity at which the program will run is decided, this stage consist in choosing the appropriate parallelism model. In parallel computing, there are mainly four parallelism models : (1) data-parallelism, (2) task-parallelism, (3) nested task- and data-parallelism, and (4) pipeline-parallelism. This choice depends on the parallelism opportunities available in the application.
- Stage 4 : it consists in choosing the programming model to map the parallelization strategy depending on the target parallel architecture.

This exploration process is very complex and requires a lot of time and effort for designers. In order to facilitate this design task, we presents the experience and the resulted guidelines for the solutions space of the parallel implementation of a representative applications : The extCED.

4.7 Experiments and Results

In this section, we evaluate the performance in term of execution time of the studied parallelization strategies applied to extCED for multicore CPU and manycore GPU. Based on the

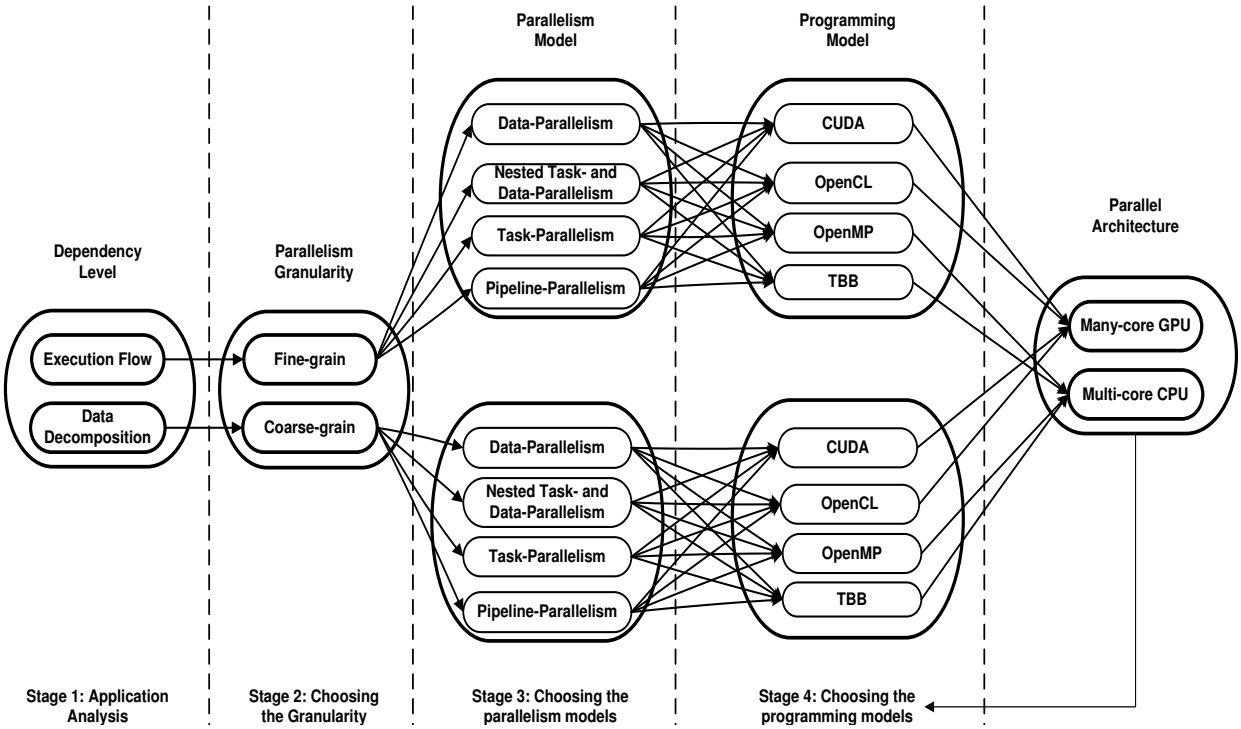


Figure 4.3 Design Challenges and Approach

experimental results, a number of conclusions are extracted that can be used as guidelines to parallelize applications presenting similar features in the future.

4.7.1 Implementation Details and Experimental Setup

Our experiments are based on an extended CED that we developed in C. We propose different parallelization strategies for the extCED. We use OpenMP 3.0 and Intel TBB as programming models for multicore CPU, and CUDA and OpenCL as programming models for manycore NVIDIA GPU. The multicore CPU target platform is a dual AMD Opteron 6128 runing at 2 GHz. The manycore GPU target platform is a NVIDIA GeForce GPU specifically a Fermi-series graphics processor GTX 480 which integrates 480 SPs distributed on 15 Streaming Multiprocessors (SMs) as 32 Streaming Processors (SPs) per SM. The running frequency of the GPU is 1.4 GHz and the assigned global memory is 1.5 GB. We use CUDA 4.8 and OpenCL 1.1 to program the GPU.

In this work, we experiment the mentioned parallelization strategies : (1) parallelization granularity which is discussed in section 4.4.1 and (2) parallelism model which is discussed

in section 4.4.2. Each parallelization strategy is implemented on two parallel platforms using two programming models for each platform : (1) multicore CPU using OpenMP and TBB and (2) manycore GPU using CUDA and OpenCL. We develop new parallel version of extCED for each experiment as it will be shown later in this section. Figure 4.4 shows these parallel versions, where each version represented as a path linking the different points from the sequential program to the target architecture. Each crossed frame in this figure represents a parallelization strategy category, and each point represents one of the studied parallelization strategy. For all experiments we capture the execution times of four programming models as two programming models for each target architecture. For multicore CPU, we use AMD Code Analyst profiling tool to collect statistics like the number of clock cycles per each as the main metric to check load balancing and execution performance, L3 cache misses and programming model runtime overhead as influential factors on performances. For manycore GPU, we use Nsight the profiling tool provided by NVIDIA to collect the execution time and the statistics about accesses to the different levels of hierarchy. In order to facilitate a clear display of the results, the execution times are normalized around a maximum value which corresponds to 100% on the y axis. The shown execution times are the average values of 50 executions. In order to study the impact of the size of processed data on the performances, we perform experiments on three image sizes : (1) 512x512, (2) 1024x1024 and (3) 2048x2048.

4.7.2 Fine-grain vs. Coarse-grain Parallelism

In this section, we capture the execution time of two different parallelization granularities : (fine-grain and coarse-grain). The experiments focus on Part 1 and Part 3 of extCED program (see Figure 4.1). Part 1 includes a number of successive image processing operations and Part 3 include dynamic behaviour given by the variable bound condition of a *while* loop which encloses 3 image processing operations. Part 2 is not considered in the experiments since it does not show any kind of parallelism.

Fine-grain vs. Coarse-grain on Multicore CPU

The results using OpenMP and TBB for Part 1 and Part 3 are shown in Figure 4.5 and Figure 4.6, respectively. The difference in performance is more significant for larger images. Moreover, the difference in execution time is more visible in Part 1 than in Part 3. Moreover, the experiments show that coarse-grain parallelism gives lower execution time than fine-grain parallelism for all image sizes and parallel structures for both OpenMP and TBB. These results are essentially due to the data cache locality factor. In fine-grain parallelism, each operation is parallelized separately and applied to the whole image. In this case, the

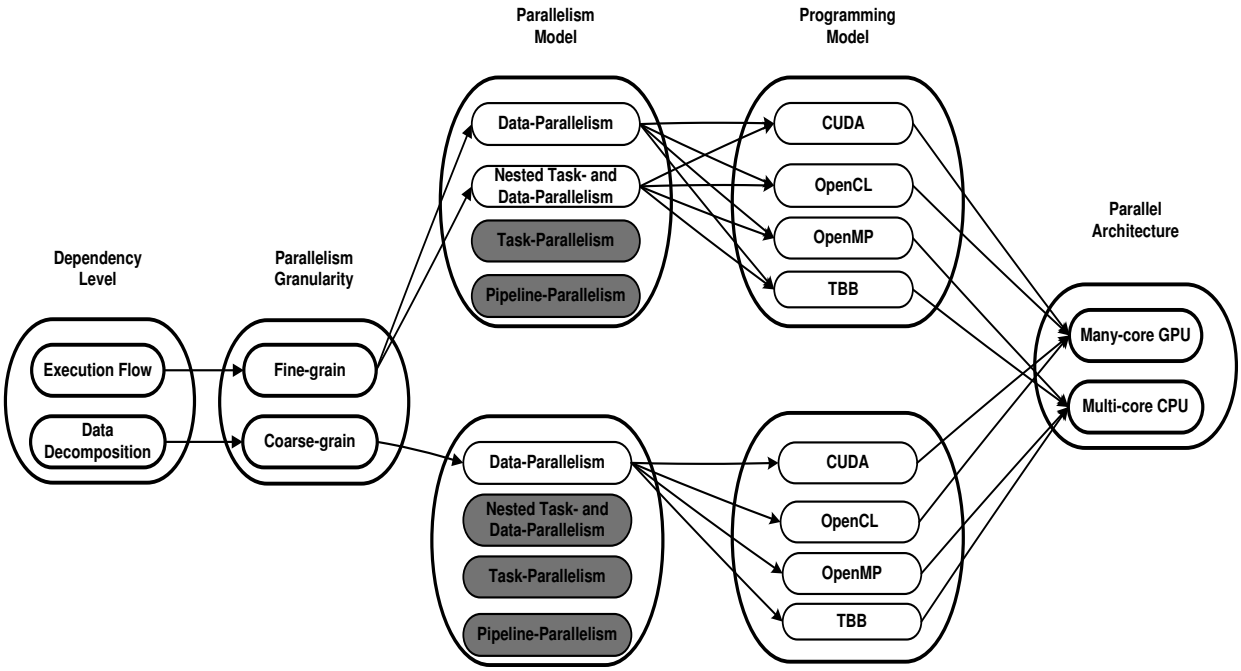
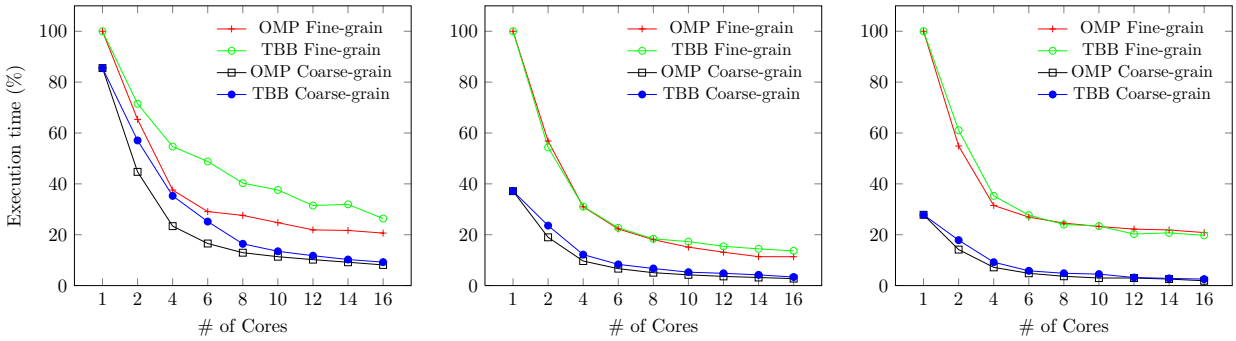


Figure 4.4 Exploration Space

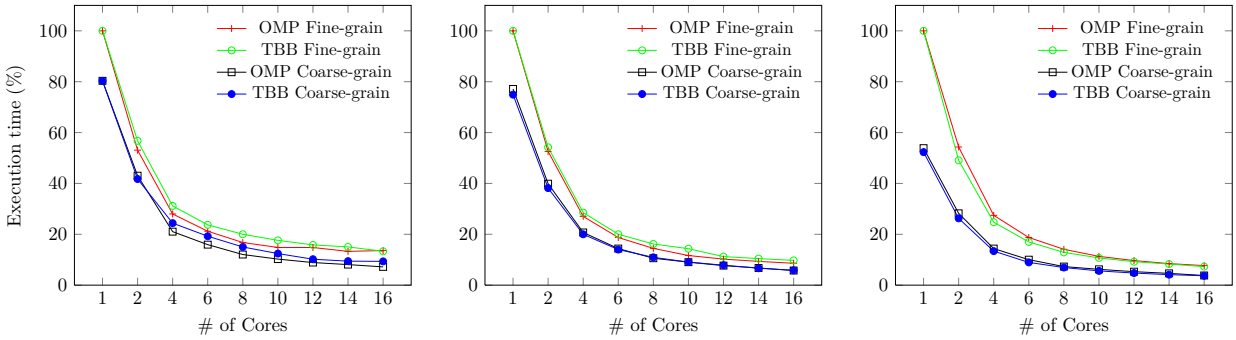
data do not remain in cache to be reused for the next operation which lead to additional clock latency to load data from main memory. However in coarse-grain parallelism, all operations are applied for each image slice. In this case, the data remain in cache for the next operation. As the intermediate data are larger, the data cache locality factor has more significant impact on execution time. This explains why Part 3 takes advantage of coarse-grain parallelism better than Part 1 since it employs a large number of 2D arrays to store intermediate results between successive operations.

As conclusion, we can claim that it is worth using coarse-grain parallelism when it is possible and especially in the case of large data toward faster execution. However, the good performance that can be reached with coarse-grain parallelism is at the price of less flexibility to exploit other forms of parallelism inside the coarsened program and the difficulty to reuse this coarsened parallel program in more complex applications. Moreover, since both OpenMP and TBB give us almost the same speedup for respectively fine-grain and coarse-grain parallelism, the developer can choose the programming model that is more familiar for him without loosing performance.



(a) 512x512 (b) 1024x1024 (c) 2048x2048

Figure 4.5 Execution Time of Section 1 on Multi-core CPU



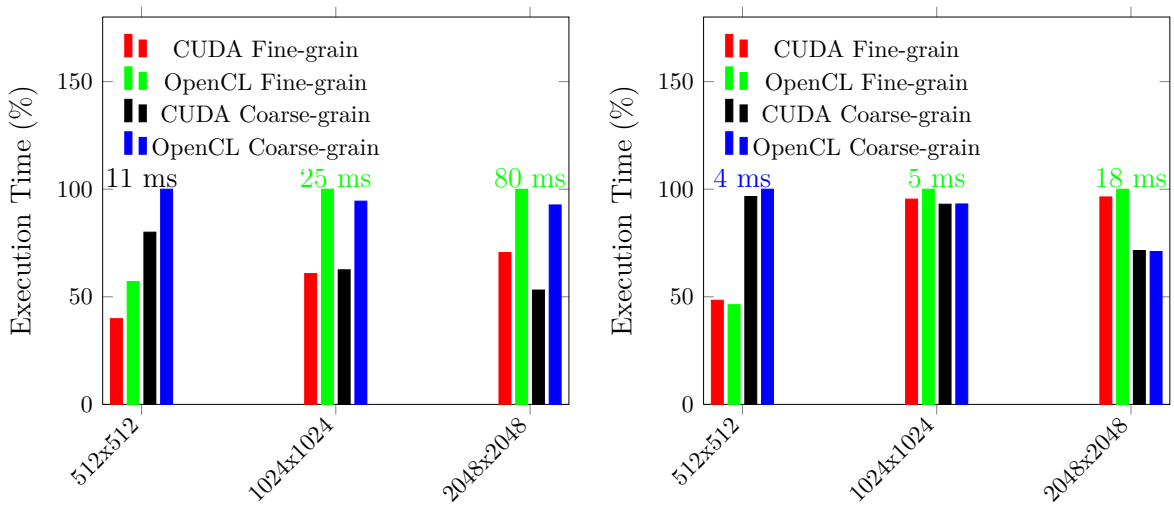
(a) 512x512 (b) 1024x1024 (c) 2048x2048

Figure 4.6 Execution Time of Section 3 on Multi-core CPU

Fine-grain vs. Coarse-grain on Manycore GPU

The results of the experiments regarding the parallelization strategies related to the granularity are shown in Figure 4.7. The execution time is represented as a percentage of a maximum execution time for each image size where each maximum is mentioned in the graphics in milliseconds. The experiments show that for small images (512x512), fine-grain parallelism performs better than coarse-grain parallelism (see Figure 4.7). This is explained by the fact that in the case of small images corse-grain parallelism created a significant smaller number of parallel threads which leads to low occupancy on the GPU and poor degree of parallelism compared to fine-grain parallelism. However, as we enlarge the image size, the performance of coarse-grain improves and outperforms the fine-grain parallelism for 2048x2048 image, since it benefits from a relatively large number of created threads and efficient use of data locality.

In term of programming models, both CUDA and OpenCL have the same behavior regarding the two granularities. While CUDA and OpenCL give almost the same execution time for processing Part 3 of extCED (see Figure 4.7(b), CUDA offers lower execution time than OpenCL for processing Part 1 of extCED (see Figure 4.7(a)). This is explained by the low performance APIs implemented in OpenCL that manage data copies between the host and the device memory. This effect is particularly visible in presence of large amount data handled by Part 1, as opposed to Part 3.



(a) Section 1

(b) Section 3

Figure 4.7 Fine-grain vs. Coarse-grain on Many-core GPU

4.7.3 Nested Task- and Data-parallelism vs. Data-parallelism

As mentioned in section 4.4.2, Gradient Calculation can be processed as two parallel tasks, where each task processes a Gradient Calculation at one direction (see Figure 4.1). Each task encapsulates loops that can be parallelized using the data parallelism trend (see Listing 4.2). Moreover, Non Maxim Suppression (see Figure 4.1) integrates the nested task and data parallelism, but with four tasks that can be processed in parallel (see Listing 4.4). Based on these parallel structures found in extCED, we perform the experiments by capturing the execution times of two different parallelism models (data-prallelism and nested task- and data-parallelism) for (1) Gradient processing and (2) Non Maxima Suppression.

Nested Task- and Data-parallelism vs. Data-parallelism on Multi-core CPU

Figure 4.8 and Figure 4.9 show the execution time of the different parallel versions of the Gradient processing and Non Maxima Suppression, respectively, for different image sizes. Based on the experimental results, we can distinguish two cases :

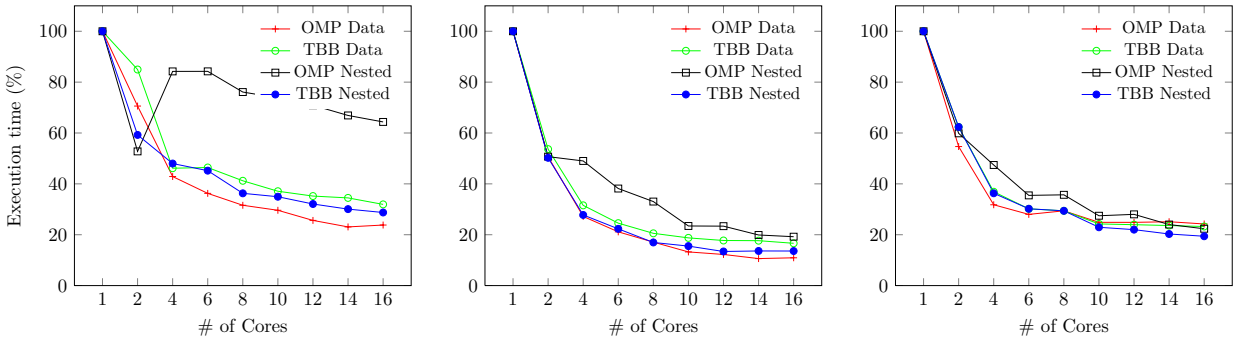
– **Case 1 (# of cores = # of parallel tasks) :** in this case the nested task- and data parallelism is rather a pure task-parallelism since each task is assigned to a separate core and no more cores are available to manage the data-parallelism present within each task. We see this case in Figure 4.8 when the number of cores is equal to two and in Figure 4.9 when the number of cores is equal to four. As first conclusion, we can claim that pure task-parallelism offers the best performances for all image sizes except for the case of Figure 4.8(c). This can be explained by the low runtime overhead introduced by the studied programming models that is required to manage pure task parallelism compared to the overhead that is required to manage a pure data-parallelism.

– **Case 2 (# of cores >\# of parallel tasks) :** Nested task- and data- parallelism with TBB offers better performances than OpenMP across the different images sizes. This is due to the nested parallelism implementation in OpenMP, which suffers from a significant runtime overhead. However OpenMP seems to be more efficient for pure data-parallelism than TBB. This is due to the data parallelism management in TBB. Moreover, we can observe significant impact of unbalanced load for the TBB implementation. This is due to the concentration of the load in one thread playing the role of a master thread, as opposed to OpenMP where the load is equally shared between processing cores.

In conclusion, we can claim that OpenMP offers better performances with data-parallelism than nested task- and data-parallelism. This is due to the additional overhead to manage nested parallelism compared to pure parallelism. However, TBB offers better performances with nested task- and data-parallelism than pure data-parallelism.

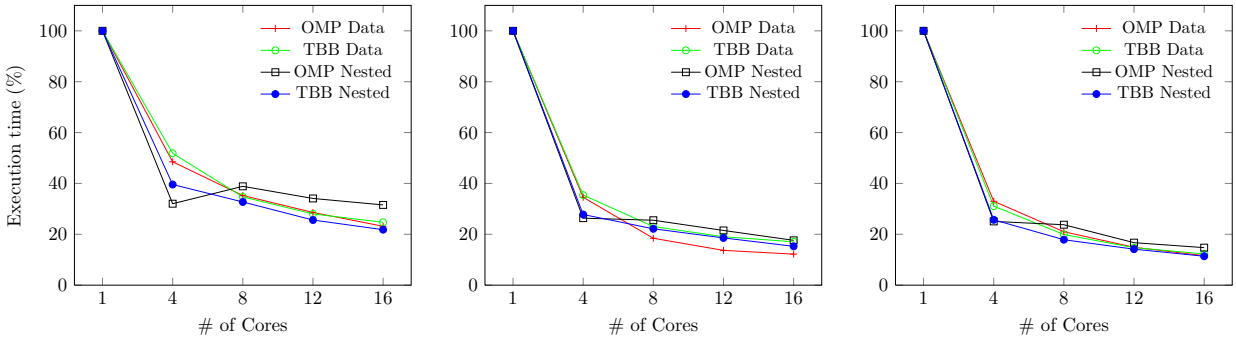
Nested Task- and Data-parallelism vs. Data-parallelism on Many-core GPU

The nested task- and data-parallelism on GPU gives almost the same performances as pure data-parallelism. This can be explained by the fact that Fermi architecture GPUs implement only one hardware queue to execute kernels. Therefore parallel kernels cannot fully take advantage of parallel execution. Rather, their execution is serialized and may overlap, but only for a short time depending on available resources (registers, memory and idle cores).



(a) 512x512 (b) 1024x1024 (c) 2048x2048

Figure 4.8 Gradient Processing Execution Time on Multi-core CPU



(a) 512x512 (b) 1024x1024 (c) 2048x2048

Figure 4.9 Non Maxima Suppression Processing Execution Time on Multi-core CPU

4.8 Conclusion

In this chapter, we have investigated several parallelization strategies used in image processing applications. The motivations behind this work are first the difficulty to reach an efficient parallelization of a sequential program in presence of large number of parallelization solutions and second the missing work comparing fairly this wide range of parallelization strategies under comparable constraints. In order to evaluate the performances of these parallelization strategies, we played the role of the developer trying to parallelize an input sequential program and we explored the different parallelization strategies. At the end of this chapter, we can draw these conclusions :

- For small computations, fine-grain performs better than coarse-grain parallelism on manycore GPU. This is due to the small number of threads generated by coarse-grain parallelization which yields to a low GPU occupancy and by consequence a large ove-

rhead on waiting requested data from memory.

- For large computations, coarse-grain parallelization outperforms significantly fine-grain parallelization on both multicore CPU and manycore GPU. This is explained by the high rate of data locality that offers coarse-grain parallelization compared to fine-grain parallelization.
- OpenMP offers better performances with data-parallelism than nested task- and data-parallelism. This is due to the additional overhead to manage nested parallelism compared to pure parallelism. However, TBB offers better performances with nested task- and data-parallelism than pure data-parallelism.

CHAPTER 5 TUNING FRAMEWORK FOR STENCIL COMPUTATION

5.1 Introduction

In order to implement computer vision and image processing applications on heterogeneous parallel platforms (HPPs), a number of parameters have to be considered. A class of image processing and computer vision applications are based on Stencil computation. This type of computation is known as memory-bound computation, so that reducing high latency memory access becomes the biggest challenge to reach high performance.
To overcome this issue, it is needed to ensure better data locality through the efficient use of low latency scratchpad shared memory (Grosser et al., 2013). However, this is not obvious for the developer, especially due to the existence of :

1. a complex application data access pattern, and ;
2. resource constraints such as the available low latency memory space and the maximum number of executed threads.

Therefore a tuning framework is mandatory to guide the developer to reach high performance. We build our tuning framework based on the guidelines acquired from the performance evaluation of the optimization techniques on modern parallel platforms provided previously in Chapter 3, and the deep exploration of the parallelization strategies provided previously in Chapter 4.

In this chapter, we present a performance tuning framework for stencil computation targeting HPPs. The emphasis is put on HPP platforms that follow host-device architectural model where the host corresponds to a multicore CPU and the device corresponds to a manycore GPU.

5.1.1 GPU-based Heterogenous Parallel Platforms

Heterogeneous parallel platforms (HPPs) are used as an appropriate choice to accelerate compute intensive applications since they offer a good balance between high computation capabilities and flexibility to handle large spectrum of applications features (Pienaar et al., 2011). Most of these platforms adopt host-device model and usually the host is a cache-based heavy multi-core CPU and the device is essentially a scratchpad memory-based light many-core CPU or GPU. HPPs may be exploited in :

- High performance computing (HPC) systems : desktops and workstations that integrate multi-core general purpose CPU (Intel or AMD) as host and a discrete many-core GPU (NVIDIA and AMD) as device,
- Portable Systems : laptops and netbooks that integrate a multicore CPU (Intel or AMD) as host and GPU cores (Intel or AMD) as device.
- Low power embedded systems : smartphones and tablets that integrate in one die a multicore CPU (ARM) as host and GPU cores (NVIDIA) as device.

5.1.2 Stencil Computation

Finite difference method used to solve partial derivative equations (PDEs) is at the core of both scientific applications such as heat diffusion, climate science, electromagnetic and fluid dynamics, and image processing applications such as image filtering and edge detection. The computational pattern which expresses the finite difference method is called stencil computation (Datta et al., 2008). In stencil computation, each point of the computed space is updated depending on the points at its neighborhood which is called stencil (see Figure 5.1).

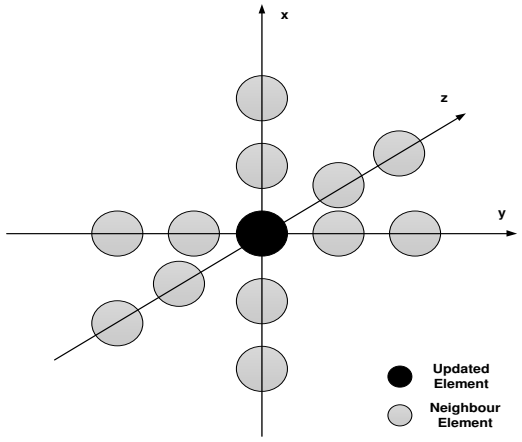


Figure 5.1 Stencil

The stencil itself may be defined by three properties :

1. stencil operations, that is, the type of operations performed on neighbor points to update the central point ;

2. stencil shape, that is, the topology of the neighbor points that are unchanged during the computation. Figure 5.2 depicts three possible situations regarding horizontal and vertical 1D, and 2D computations (namely a, b and c in the figure), and ;
3. stencil space, that is, the space of points that need to be updated. In general, this space is a regular structured multi-dimensional grid.

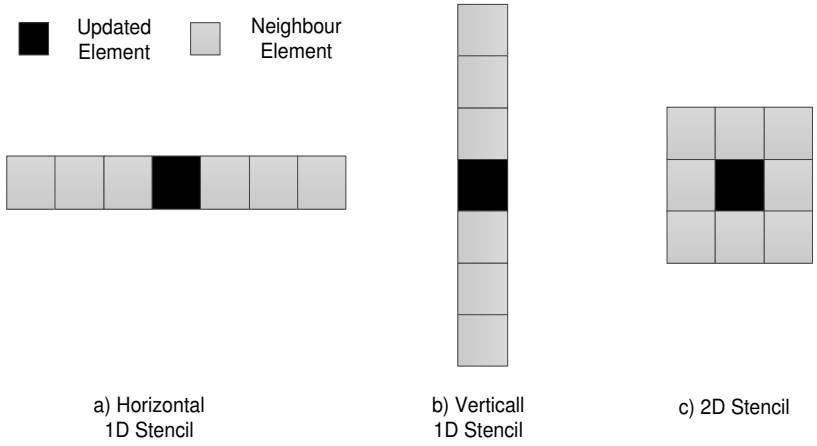


Figure 5.2 Stencil Shapes

The stencil computation may take several forms such as :

- **single stencil computation**, where a single stencil is applied only once on the stencil space. For example, we have image convolution on rows ;
- **multiple stencils computation**, where a sequence of different stencils is applied on the stencil space. The Canny Edge Detection is an example of this form, and ;
- **iterative stencils computation**, where one or several stencils are applied on the stencil space repeatedly and the number of iterations may be known at compile time. In this case, we have simulations with fixed number of time steps, or unknown at compile time that depend on a certain condition. As example we have image shape refining.

It is important to highlight that, in this work, we focus on the **multiple stencils computation**. However, our approach may be also be applied for iterative stencils computations.

5.1.3 Parallelization Strategies for Stencil Computation

When implementing stencil computations on GPU-based HPP platforms, many considerations have to be taken into account. They vary according to the complexity of the target platform and the particular properties of stencil computations. These computations are known as memory bound showing a low compute intensity (the ratio of computing operations by the memory accesses). Thus, the main emphasis is put on improving data access time by improving the data locality. This is guaranteed by the use of low latency memory sucha as the scratchpad shared memory, as much as possible. Two techniques are proposed in literature to improve the data locality : *spatial tiling* and *temporal tiling* (Datta et al., 2008).
Stencil computations are applied on a large amount of data that usually surpasses the number of cores, and even the number of threads available on the target hardware platforms. As a consequence, to achieve an improved implementation with optimal locality, the spatial tiling technique is commonly used.

The tiling technique consists in splitting the data or the stencil space into tiles. Each tile is then processed by one or a group of threads. In GPU-based platform, each tile is loaded once in the shared memory. Then, it is processed iteratively by a group of threads sharing this tile. This makes the data access to each element of the tile very fast.
However, by dividing the data into tiles, the *border dependency* problem arises. The elements at the borders of each tile need their neighbors to be updated. Since these neighbors are belonging to a different tile, the access to this data will imply a considerable overhead due to extra communications and synchronizations.
To overcome this overhead, an approach named *overlapped tiling* (Krishnamoorthy et al., 2007) is adopted. This approach consists in augmenting each tile with extra elements called *halos*, which are loaded and re-computed twice in two neighbor tiles according to one dimension. Nevertheless, this implies re-loading and re-computation overhead that is proportional to the number of halos and the halo size.
In this case, the biggest challenge is to reduce the number of halos by choosing the appropriate tile layout. An example illustrated in Figure 5.3 shows that depending on the tiling, both the number and the size of needed halos will change. The tiling of the stencil space shown in (Figure 5.3-a) may follow two different options : the *tiling 1*, depicted in (Figure 5.3–b) or the *tiling 2*, shown in (Figure 5.3–c), where the halos are in gray color. It is possible to observe that *Tiling 1* involves a larger number than *tiling 2*. In *tiling 1* we have overlapped tiling (represented with dashed lines in the figure), since the dependencies are involved at both left and right side of the tile.

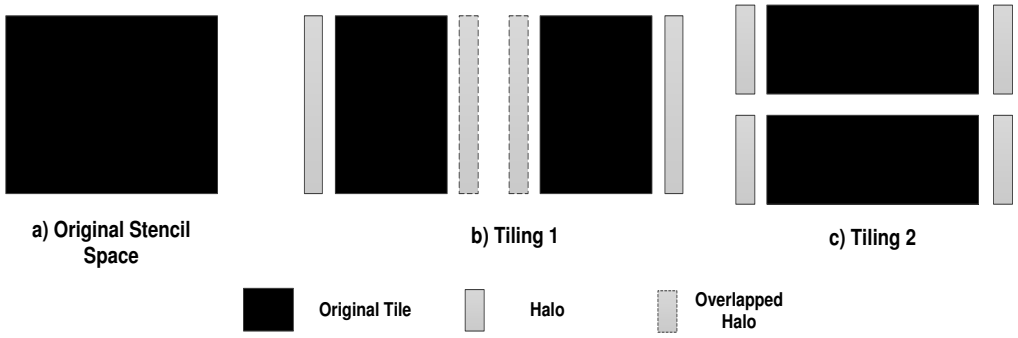


Figure 5.3 Tiling Variants

Apart from spatial tiling, the *temporal tiling* technique is used for the iterative and multiple stencil computations. Since they involve many intermediate data, we need to increase the data reuse. In a naive implementation, each time we update an intermediate data, we would access the high latency memory (device memory). In an optimized implementation, we will keep the intermediate data in the low latency memory for as long as they are needed by the computation.

In a GPU implementation, each single stencil computation is usually implemented as a separate kernel. Since the shared memory does not preserve the data between two kernels launches, multiple stencils have to be fused in one stencil to be launched only once in the case of temporal tiling.

However, by fusing multiple stencils, we have to deal with dependency at borders. To do that, we need to load the appropriate halos and find the number of allocated data and their size. Nevertheless, this definition depends on their lifetimes during the fused stencil computation. A non-optimized fusion may affect significantly the performance when the shared memory is inefficiently used. Also, performance is affected if both loading and re-computation overheads become more significant when compared to the efficient computation. Moreover, by allocating a large-sized data in the shared memory, the occupancy rate [1] will be reduced and so the concurrency degree.

1. Occupancy rate is defined as the ratio of the number of allocated threads by the limit allowed by each streaming multiprocessor (SM)

5.1.4 Contribution

In order to optimize the implementation of the existing techniques on GPU-based platforms, the developer is facing three problems :

1. What are the stencils that may be fused to reduce access to high latency memory by keeping the effective data the low latency memory as long as possible ?
2. How to find the appropriate tile size and tile shape that may fit with the low latency memory and which provides optimal performances ?
3. What is the best mapping of the computation load onto the threads hierarchy (grids and blocks of threads) ?

To guide the developer to answer to these questions, we define and implement a performance tuning framework based on a step-wise methodology which is detailed in the remaining section.

The rest of the chapter is organized as follows. Section 5.2 surveys the state-of-the-art on improving the implementation of stencil computation on parallel platforms. Section 5.3 describes in details our proposed performance tuning framework for stencil computation running on GPU-based HPP platform. Section 5.4 provides the experimental results applied on two image processing applications and provides comparison between a basic implementation and an improved one using the proposed framework. Section 5.5 concludes this chapter by a summary of the achieved work.

5.2 Related Work

Stencil computation is a well-known class of computations that is used in various domains ranging from physics simulations to image processing. A lot of effort has been spent to improve both productivity and performance of stencil computations for a wide range of target hardware platforms and in particular GPU-based platforms. This effort takes various forms such as :

1. **improving stencil computation runtime**, by elaborating algorithmic and coding optimizations that may be implemented in specific compilers or as templates in specialized libraries,
2. **performance tuning tools**, by developing tuning tools, and/or ;
3. **evaluation of stencil computation implementations**, by elaborating specialized high-level frameworks based on the definition of new Domain Specific Languages (DSL) for stencil computations.

Following, a discussion of related work classified according to these mentioned forms classified according to the above mentioned forms.

5.2.1 Related Work on Improving Stencil Computation Runtime

Several work improve the stencil computation runtime by proposing algorithmic and optimizations such as cache-oblivious algorithms, space and temporal blocking via tiling, overlapped tiling and register blocking. The work presented in (Tang et al., 2011) proposes a set of code optimizations implemented in a domain specific compiler called Pochoir. Many approaches have focused on different levels of blocking to improve cache locality on both multicore CPU and GPU architectures. K. Datta et al. studied and evaluated several optimization such as array padding. multi-level blocking, loop unrolling and reordering for stencil computation on a wide variety of hardware architectures (Datta et al., 2008).

5.2.2 Related Work on Performance Tuning Tools

There are studies that focus in developing tuning tools that target the optimization of one or both of these metrics : the tiling size, the halo size, the compute intensity and the threads blocks size for GPU. J. Meng et al. developed an analytical performance model to automatically select the halo size that gives the best speedup on GPU (Meng and Skadron, 2009). However, this model is limited to iterative stencil loops where only one stencil is involved in the whole program and, thus, not applicable to more complex code involving multiple stencils. This makes their model very simple and not applicable to more complex code involving multiple stencils. In addition, they suppose that the threads blocks are isotropic which narrows considerably the search space. On the other hand, (Wu et al., 2012) and (Tabik et al., 2014) deal with complex iterative multiple stencils where they study the impact of several combinations of kernels fusion/fission on compute intensity for GPU via an exhaustive search. However, they do not provide an analytical model to guide the search process.

5.2.3 Evaluation of Stencil Computation Implementations

Some other contributions are focused on evaluating the suitability of target platforms while running stencil computations. Specific compute capabilities of each platform were investigated and some effective implementation strategies were developed to deliver the best possible performances. In (Eberhart et al., 2014), the authors optimize the implementation of stencil computations on Accelerated Processing Units (APU) by proposing a hybrid approach. The main idea is to assign different parts of the stencil computation to different APU processors.

Since, border treatments as source of high cost of computation and memory divergence, they are assigned to CPU whereas the regular computation is assigned to the GPU part. The authors in (Calandra et al., 2013) compare the computation performance of two successive families of both discrete GPUs and integrated GPUs while running stencil code. In their paper, they evaluate the impact of Peripheral Component Interconnect (PCI) express on performances. In the same direction, the authors in (Lutz et al., 2013) evaluate the impact of PCI express on performances in the case of multi-GPU platform.

5.2.4 Positioning Our Approach

In summary, several approaches have been previously proposed for improving stencil computation performance and productivity. Complier-based approaches propose conservative tuning performance. On the other hand, implementations evaluations propose some explorations but they are not based on well defined tuning frameworks. Finally few approaches propose tuning frameworks but are limited to iterative stencil computation. Comparing with these contributions, the main strengths of the proposed approach are :

1. as opposed to conservative approaches, our framework is based on an **analytical model** taking into account the characteristics of the application as well as the characteristics of the targeted architecture, and ;
2. by considering **multiple stencils computation**, more complex applications may be optimized using our framework.

5.3 The Proposed Framework

In this section, we describe our proposed performance tuning framework for stencil computation running on GPU-based HPP platform. The proposed framework covers a large set of applications following stencil computation pattern, that is, not only image processing applications. Our methodology to build the proposed framework follows four basic steps, depicted in Figure 5.4).
The methodology steps' are :

- **Step 1 : formulation for stencil computation running on HPP.** This formulation allows abstracting implementation details while keeping the relevant aspects having an impact on performance. The high level representation can then be projected on a performance model to predict the performance. Thus, the developer does not have to implement a full functional code in CUDA or OpenCL for every parallel version to check its performance. Details about the formulation and illustrative examples of high-level

representations are provided in Section 5.3.1.

- **Step 2 : generates all possible parameters configurations depending on the target platform.** For each configuration, the resource usage (shared memory usage, thread occupancy and the number of resident blocks) and the number of iterations needed to compute the full stencil space. The outputs of this step are injected as inputs of step 4 in order to select the set of optimal configurations with best performances. Details about this step are provided in Section 5.3.2
- **Step 3 : determines the influential parameters on performance metrics and identify those that can be controlled by the developer to tune performances.** For this purpose, we profile three data access patterns used in three stencil shapes which are the basic stencils used in a majority of image processing applications. Details about this step are provided in Section 5.3.4
- **Step 4 : we provide at this step a performance model for stencil computation.** This model takes as input the resource usage and the computation load provided by step 2, and the impact of a set of controlled parameters on performance metrics that are provided by step 3. This performance model allows to assign to each implementation configuration a computational cost that is used to determine the highest performance configurations. Details about this step are provided in Section 5.3.5

These steps will be detailed during the next sections.

5.3.1 Formulation of Stencil Computation Running on HPP (Step 1)

In this section, we define the first step of our methodology. We provide a formulation for a stencil-based program that runs on a HPP based on host-device schema. We analyze the particular case of a CPU-GPU platform, as depicted in Figure 5.5.
to achieve our goal, we first define a formulation for a general program running on HPP. Second, we define a particular formulation for stencil computation.

Assumptions and Simplifications

We consider in this work 2D grids since we are essentially targeting image processing and computer vision applications. However, the proposed framework is still applicable for 3D grids. In our formulation, we consider only data transfer between host and device. We omit the representation of host program since we focus essentially on processing on GPU. A particular emphasis is put on multiple stencil computations running on such platforms. As assumption, we suppose that the studied stencils are symmetric and grids are regular Cartesian grids. The

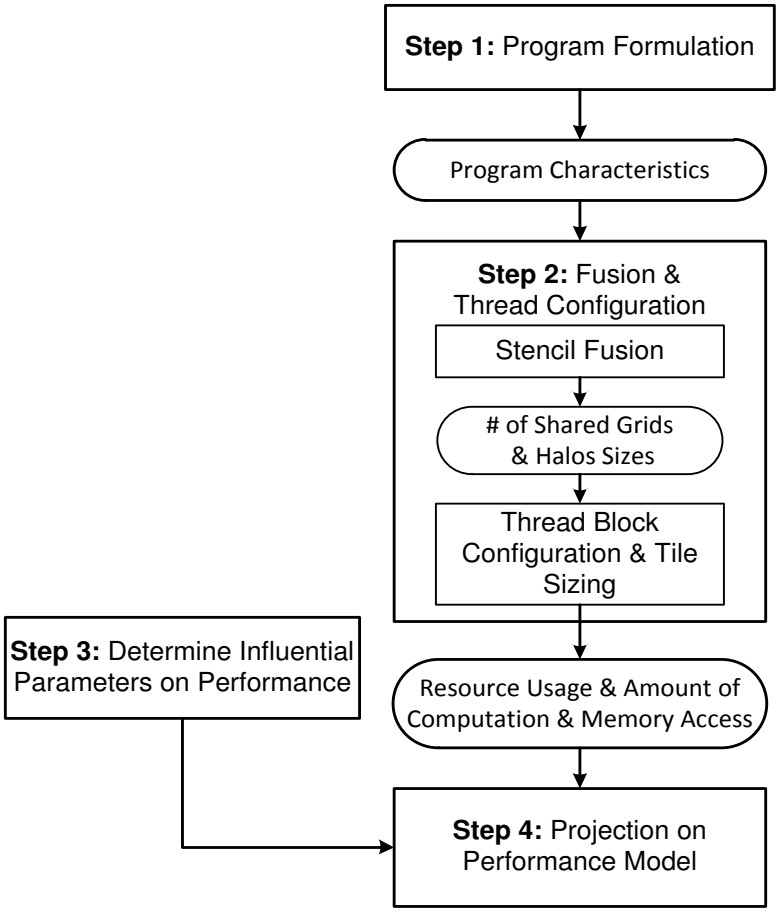


Figure 5.4 Proposed Methodology

formulation is a single stencil based formulation. A single stencil is a stencil that performs a single processing and produces only one output grid but may consume more than one input grid. The stencils involving more than one processing and more than one output grid have to be decomposed into separate single stencils so they can be supported by the proposed formulation.

Formulation of Program Running on HPP

The main formulation terms and their corresponding descriptions are summarized in Table 5.1.

At platform-level, we consider two main types of memory $MemType$: (i) high latency global memory denoted by $GlobalMem$ and (ii) low latency scratchpad shared memory denoted by

Table 5.1 Summary of Main Formulation Terms and Notations

Symbol	**Description**
$\rightarrow$	Sequence of actions
:	Type
=	Definition
Term	**Description**
DTransfer	Data transfer between host memory and global memory
DAccess	Data access from/to global memory or shared memory
Proc	Processing which includes computation and data access
Comp	Computation without data access consideration
Sync	Synchronization mechanism

$ShMem$. Memory type $MemType$ is defined by :

$$MemType : GlobalMem \mid ShMem \tag{5.1}$$

In our formulation, a program running on HPP is represented as a skeleton expressed as a sequence of processing denoted by *Proc*, data transfer denoted by *DTransfer* and synchronization denoted by *Sync* :

$$Program \rightarrow Proc \quad DTransfer \quad Sync \tag{5.2}$$

A Processing is further expressed as a sequence of data access denoted by $DAccess$, operation denoted by Op and synchronization denoted by $Sync$:

$$Proc \rightarrow DAccess \quad Op \quad Sync \tag{5.3}$$

$Proc$ may have different processing types. We focus in particular on stencil processing, detailed in Section 5.3.1 :

$$ProcType : StencilProc \mid Other \tag{5.4}$$

As for synchronization mechanisms *Sync*, there are two types :

1. **barrier-based synchronization** - *BarrierSync*, that denotes the synchronization between threads belonging to the same thread block, and ;
2. **event-based synchronization** - *EventSync*, that denotes the synchronization between different streams.

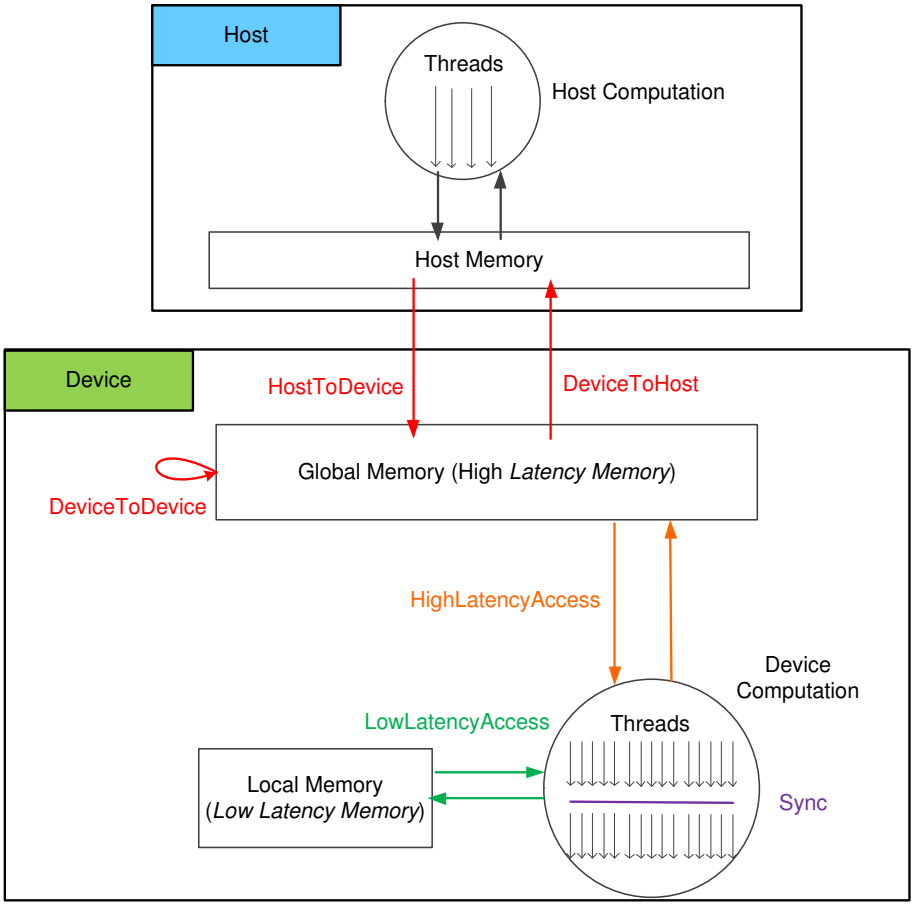


Figure 5.5 Abstracted Model of Heterogeneous Parallel Platform

Thus, the definition of the type of synchronization mechanism is as follows :

$$SyncType : BarrierSync \mid EventSync \tag{5.5}$$

Still, *DTransfer* can be classified in the following types :

1. *HostToDevice* which denotes data transfer from host memory to device memory ;
2. *DeviceToHost* which denotes data transfer from device memory to host memory, and ;
3. *Device* which denotes within device memory.

Thus, the definition for $DTransferType$ is as follows :

$$DTransferType : HostToDevice \mid DeviceToHost \mid DeviceToDevice \tag{5.6}$$

Dtransfer is also defined by the transfer type, denoted by $DtransferType$ and the parame-

ters :

1. $\#ELem$ to denote the number of transfered elements, and ;
2. $ElemSize$ to denote the size in Bytes of one depending on its type : ($char$, $integer$, $float$, etc,...).

Thus, the definitions of the transfer type are as follows :

$$DTransferDef = DTransferType(\#Elem, ElemSize) \tag{5.7}$$

A data access type *DAccessType* could be a load or store operation from/to *GlobalMem* or *ShMem*. Hence, the types of data access : *DAccessType* is defined by :

$$DAccessType : Load \mid Store \tag{5.8}$$

A data access definition $DAccessDef$ is expressed by $DAccessType$ and two parameters : (i) the number of accessed elements $\#Elem$ and (ii) the element size $ElemSize$:

$$DAccessDef = DAccessType(MemType, \#Elem, ElemSize) \tag{5.9}$$

We consider three types of operations : (i) integer : I (ii) single precision : SP and (iii) double precision : DP. Operation type : $OpType$ is expressed by :

$$OpType : I \mid SP \mid DP \tag{5.10}$$

Op is defined by the type of operation $OpType$ and the number of operations : $\#Op$:

$$OpDef = OpType(\#Op) \tag{5.11}$$

Example of Program Running on HPP

This section illustrates the presented formulation using as example a simple image blending application (see Figure 5.6). An image is represented as a 2D grid of pixels of size $IM = IM_y \times IM_x$ where IM_y and IM_x are respectively the height and the width of the image. Each pixel is located in the grid by its coordinates y and x. The blending superposes two input images : $inImg1$ and $inImg2$ assigned with different weights by varying a factor α. The operation of blending is given by this equation :

$$outImg(y, x) = (1 - \alpha) \times inImg1(y, x) + \alpha \times inImg2(y, x) \tag{5.12}$$

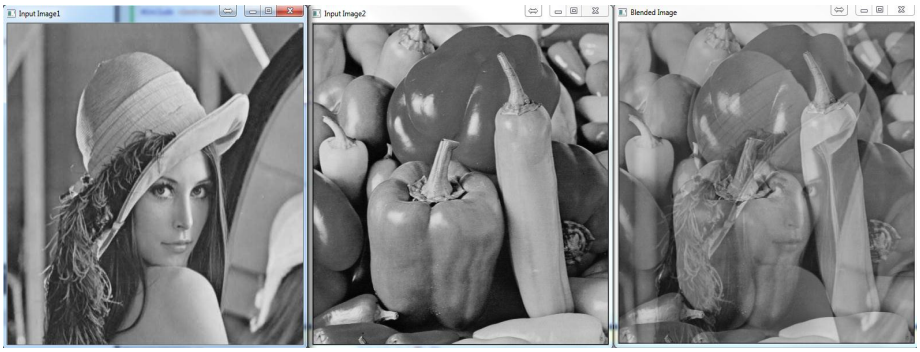


Figure 5.6 Image Blending

Following, Listing 5.1 shows the code skeleton of two gray scale images blending running on HPP expressed using the proposed formulation terms. The input images of size $IM_y \times IM_x$ are transferred from $HostMem$ to $GlobalMem$. The images stored in $GlobalMem$ are divided into tiles of size ($TL = TL_y \times TL_x$) over the parallel thread blocks. Each thread block Blk iterates over its assigned number of tiles denoted by $\#TL$. At the last stage, the resultant tile is stored to $GlobalMem$.

Listing 5.1 Code Skeleton of Blending Program

```
DTransfer:HostToDevice(IM, 4);
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    DAccess:Load(GlobalMem, TL, 4);
    DAccess:Load(GlobalMem, TL, 4);
    Op:SP(TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

Formulation of Stencil-based Processing

After defining a program running on HPP, we define, in this section, a particular type of processing *StencilProc* which is the stencil-based processing. *StencilProc* is defined by the following sequence :

$$StencilProc \rightarrow DAccess(Stencil) \quad Op(Stencil) \quad Sync(Stencil) \tag{5.13}$$

In our framework, we define a simple stencil by a stencil that consumes one or multiple grids and produces only one grid. In typical stencil processing, each input grid is divided into tiles of size TL over a number of thread blocks $\#Blk$. In order to accelerate processing via accessing low latency memory $ShMem$, each tile is loaded to $ShMem$ to be processed by a given thread block.
Since each thread block has a view of only a part of the data restricted to that loaded on the local address space allocated for it, the problem of dependency at borders will arise. To handle efficiently the dependencies at the borders of tiles, a well-known technique called *overlapped tiling* (Krishnamoorthy et al., 2007) is employed.
This techniques consists in augmenting each tile with a extra halo elements H as shown in Figure 5.7.
In this case, we define each tile size ($TL = TL_y \times TL_x$) and each halo size ($H = H_y \times H_x$), so the augmented tile of size TL' loaded into the shared memory is defined by :

$$TL' = (TL_y + 2 \times H_y) \times (TL_x + 2 \times H_x) \tag{5.14}$$

A stencil is defined in our formulation by its identifier $StencilID$ and the parameters :

1. a set of pairs of input grid $inGrid_i$ and its corresponding halo H_i ;
2. an output grid $outGrid$, and ;
3. a transition function $TransFunc$ shown in Expression 5.15.

In stencil computation, each element of the output grid is calculated depending on the elements' values at its current position and its neighborhood of an input grid by applying a transition function $TransFunc$. For each dimension dim, the size of the stencil is calculated as $(2 \times H_{dim} + 1)$ where H is the halo added to the input grid to compute the elements of the output grid.

$$StencilDef = StencilID(\bigcup_{i}^{\#inGrid} \{(inGrid_i, H_i)\}, outGrid, TransFunc) \tag{5.15}$$

$Grid$ defines an N-dimensional Cartesian grid when has its identifier $GridID$, its number of dimensions $\#Dim$, its size $Size = \prod_{dim}^{\#Dim} Size_{dim}$ and the size of each element in bytes $ElemSize$:

$$GridDef = GridID(\#Dim,\ Size,\ ElemSize) \tag{5.16}$$

TransFunc defines the stencil operations Op on one grid element and is expressed by :

$$TransFuncDef = OpType(\#Op) \tag{5.17}$$

The number of operations, denoted by $\#Op$ is given by the following expression :

$$\#Op = \sum_{i}^{\#inGrids} \left(\prod_{dim}^{\#Dim} ((2 \times H_{i,dim} + 1) \right) \tag{5.18}$$

Example of Stencil-based Processing

As example of stencil-based processing, we consider a simple image convolution on rows. The input Image is of size $IM = IM_y \times IM_x$ where IM_y and IM_x are respectively the height and the width of the image. The operation of convolution is performed on each pixel of the image by applying an 1D filter of aperture size of $(2 \times radius + 1)$ centered on that pixel. The result of convolution is stored at the centered pixel in the output image as shown previously in Figure 5.7). The convolution operation is given by :

$$outImg(y, x) = \sum_{-radius}^{radius} inImg(y, x + k) \times filter(k + radius) \tag{5.19}$$

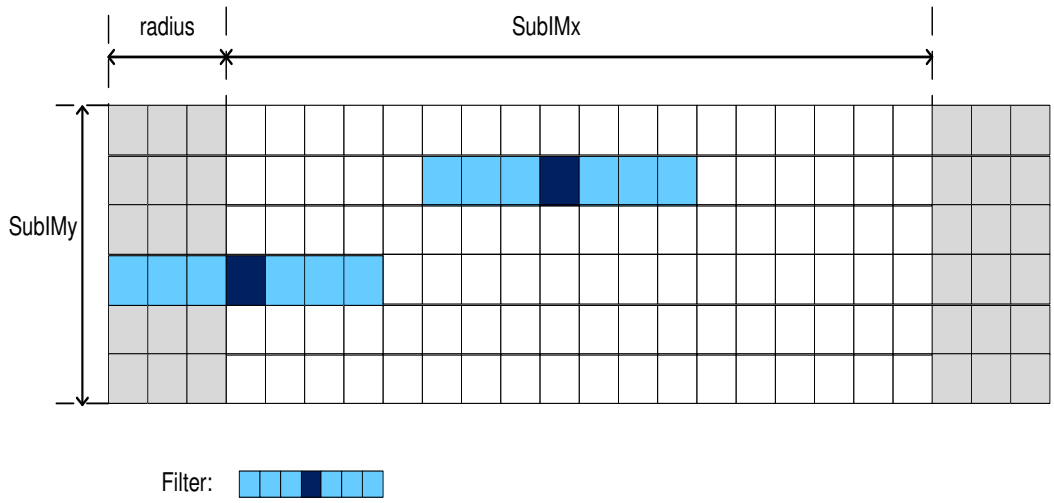


Figure 5.7 Convolution on Rows

The convolution processing expressed in the defined formulation for stencil computation is given in Expression 5.20. $Conv$ is the transition function performing the convolution operation which is defined by Expression 5.21.

$$Stencil(Convolution) = StencilConv((inImg, radius), outImg, Conv) \tag{5.20}$$

$$Conv = SP(\#Op(Conv)) \tag{5.21}$$

$$\#Op(Conv) = 2 \times radius + 1 \tag{5.22}$$

In the following, we present the code skeletons of two versions :

1. the first version without shared memory usage (see Listing 5.2),
2. the second version using the shared memory (see Listing 5.3).

For both versions, the input image of size IM is transferred from $HostMem$ to $GlobalMem$. The image stored in $GlobalMem$ is divided into tiles of size ($TL = TL_y \times TL_x$) over the parallel thread blocks. Each thread block Blk iterates over its assigned number of tiles denoted by $\#TL_{Blk}$. For convolution with use of shared memory, each tile is augmented with an extra halo : ($2 \times radius$) and stored in $ShMem$. The size of the augmented tile is denoted by TL' is given by :

$$TL' = TL_y \times (TL_x + 2 \times radius) \tag{5.23}$$

Also, in both versions, the convolution is applied on each tile element by performing ($2 \times radius + 1$) loads and single precision operations. The output tile is stored to $GlobalMem$ and all the output tiles forming the output image $outImg$ are transferred back to the host memory.

Listing 5.2 Code Skeleton of Convolution without Shared Memory

```
W = 2*radius+1;
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    DAccess:Load(GlobalMem, W × TL, 4);
    Op:SP(W × TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

Listing 5.3 Code Skeleton of Convolution with Shared Memory

```
W = 2*radius+1;
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    DAccess:Load(GlobalMem, TL', 4);
    DAccess:Store(ShMem, TL', 4);
    Sync:BarrierSync
    DAccess:Load(ShMem, W × TL, 4);
    Op:SP(W × TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

5.3.2 Fusion and Thread Configuration (Step 2)

In this section, we define the second step of our methodology. The program characteristics, such as the data grid size, its dimensions, and the halo size are the main input. This step is divided in two sub-steps :

1. **sub-step 1 :** for each tile placed in the shared memory the total halo size needed to be added to avoid any extra data exchange with global memory. This step is performed for each fusion combination defined by the developer and depends on the halo size defined for each stencil during the stencil formulation at step 1. At the end of this sub-step the total halo size for each tile is extracted and injected to the second sub-step, and ;
2. **sub-step 2 :** it takes as input both the number of tiles placed in shared memory and the halo size computed previously. It generates a number of possible configurations under the architecture constraints. For each generated configuration, a set of resource usage is also provided.

Stencil Fusion Problem

In iterative stencil computation, *stencil fusion* (also called *time tiling* is an optimization technique, which consists in grouping a number of stencils together. This technique allows better data locality, especially in the case of GPU. Since the data access to global memory has an expensive computational cost, the idea is to fuse several stencils and let the intermediate data resident in the shared memory for as long as possible. However, the fusion has some side effects, such as extra data load and re-computation overheads, which are necessary to

avoid data updates via the access to global memory.

The implementation of fusion on GPU is not an easy task and in particular with multiple stencil shapes and data grids. The developer has the challenge of determining the size and the shape of halos that will be added to each involved grid in the computation of the fused stencils. Also, another challenge is to figure out which stencils should be fused. An example of different stencil combinations is shown in Figure 5.8.

In order to help developer to implement the best stencil fusion combination, we provide a tool to compute the appropriate halo size and shape to be added to each involved data grid for a selected fused stencils. The algorithm implemented at this stage can be seen in Algorithm 1 and is described below. The output of this tool serves as input for a second tool (see Section 5.3.2) which provides all possible thread mapping configurations and their corresponding GPU resource usage for the selected fused stencils. The second tool is described in Section 5.3.2. At the final stage, an analytical performance model which is described in Section 5.3.5 is used to generate the possible configurations according to their computational costs.

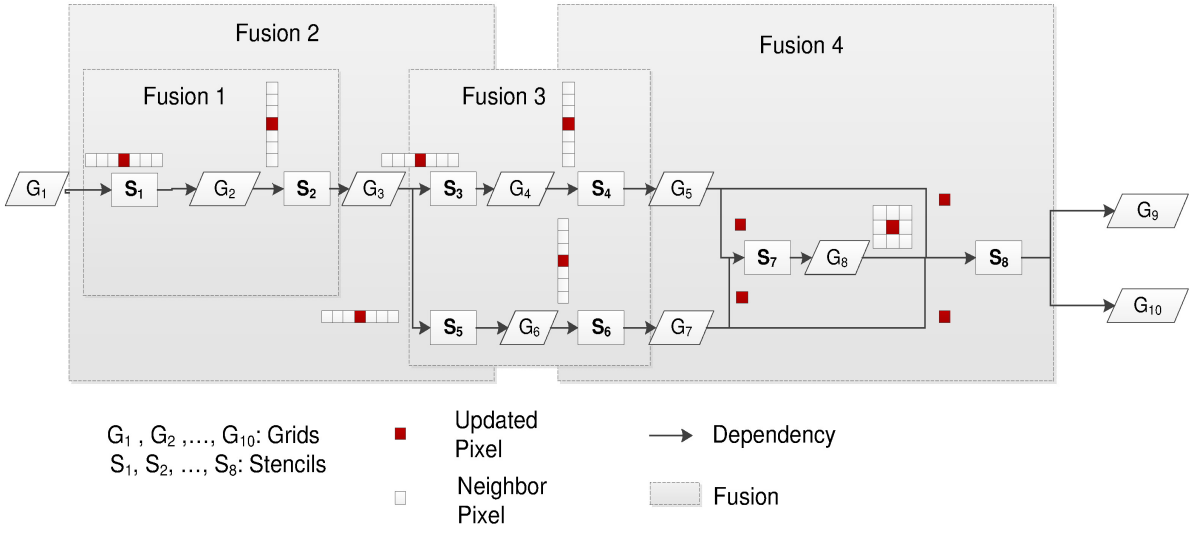


Figure 5.8 Fusion Combinations

A stencil based application shown as input application in Figure 5.9 could be represented as a weighted direct acyclic graph (DAG) where vertices represent the data grids G_i and edges represent the stencils and their weights represent the halo size H to be added for the stencil computation. Each halo is denoted by $H(inGrid, outGrid)$ which defines the halo to

be added to *inGrid* to compute *outGrid*. When we apply fusion to a group of stencils, all grids, except output grids, which are represented as black vertices, are loaded in the shared memory. All grids loaded in the shared memory denoted by *shGrids* are represented as white vertices.

The total halo size to be added to grids loaded in shared memory to update the output grids needs to be determined in order to solve the dependency problem and to avoid any extra data exchange with the global memory. For this purpose, the initial graph is treated as two separate graphs, one graph for each output grid.

Then, for each graph we determine the halo size to be added to each *shGrid* in order to compute the output grid for that graph. A halo with size $H'(i,j)$ is added to $shGrid_i$ in order to compute $outGrid_j$. To find this size, the problem is reduced to a longest path problem. In this case, $H'(i,j)$ is the longest path length separating the vertices $shGrid_i$ and $outGrid_j$. At the final stage, the total halo size to be added to each $shGrid_i$ in order to compute all the output grids denoted by H'_i is the maximum size among all the halo sizes to be added to compute each output grid as shown in Equation 5.24.

$$H'_i = \max_j(H'(i,j)), \forall i \in shGrids, \forall j \in outGrids \tag{5.24}$$

Algorithm 1: Halo Size Computation Algorithm

Input : Weighted DAG of Fused Stencils
Output: H'_i : Total Halo Size for each $shGrid_i$

```
foreach shGrid_i do
    Intialize H'_i = 0
end
foreach outGrid_j do
    foreach shGrid_i do
        Compute H'(i,j) = LongestPath(shGrid_i, outGrid_j)
        Update H'_i = max(H'_i, H'(i,j))
    end
end
```

Example of Stencil Fusion

To illustrate our approach, we apply fusion on an image processing application : the Gaussian Blur filter operation which consists of two successive separate 1D convolution operations applied on rows and on columns. Gaussian Blur could be represented as two successive stencils (see Figure 5.10). The first stencil is a 1D horizontal convolution involving a halo

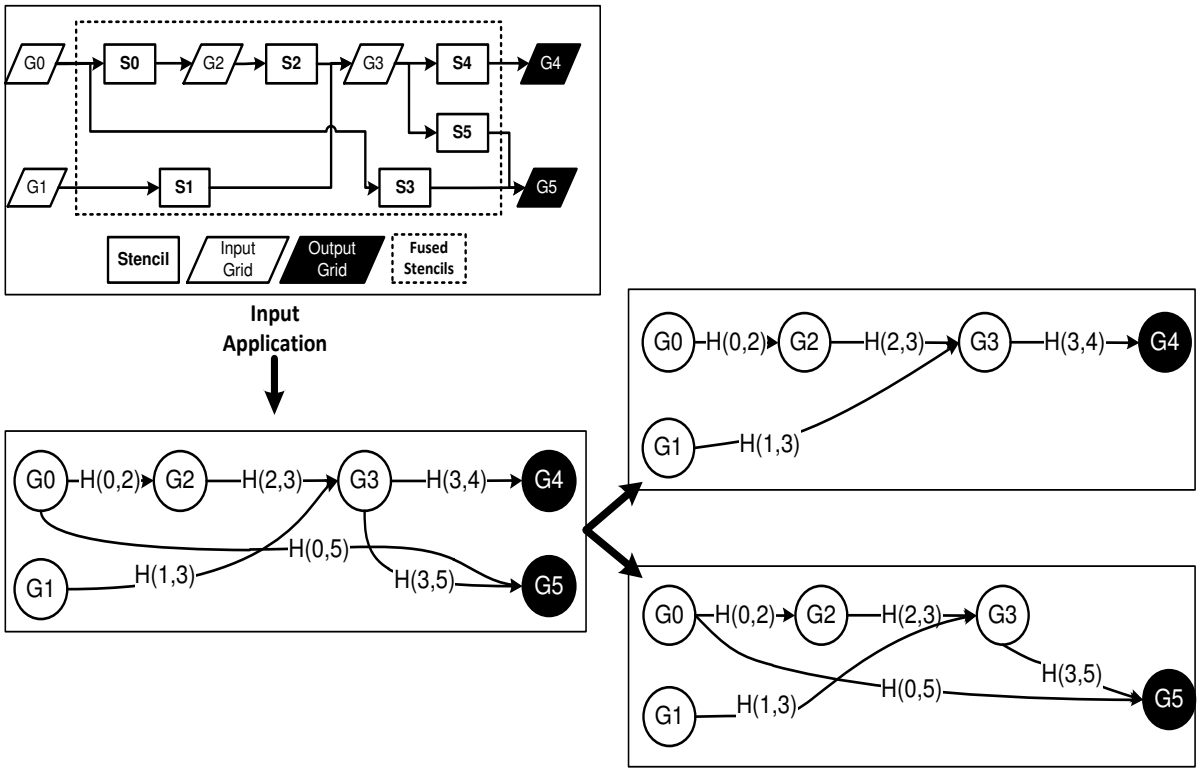


Figure 5.9 Representation of Fused Stencils

of size r and the second stencil is a 1D Vertical convolution involving a halo of size r. The representation of the two stencils using our formulation are shown in Table 5.2. The code skeleton of the fused Gaussian Blur is illustrated in Listing 5.4 where the halos to be added to each grid in shared memory G_1 and G_2 are given by :

$$\begin{cases} H'_2 = H(2,3) = r \times 1 \\ H'_1 = H(1,2) + H(2,3) = r \times r \end{cases} \tag{5.25}$$

Listing 5.4 Code Skeleton of Fused Gaussian Blur

```
H'_1 = (r, r);
H'_2 = (r, 0);
TL'_1 = (TL_y + H'_{1,y}) × (TL_x + H'_{1,x});
TL'_2 = (TL_y + H'_{2,y}) × (TL_x + H'_{2,x});
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    \\load Input Tile in ShMem
    DAccess:Load(GlobalMem, TL'_1, 4);
    DAccess:Store(ShMem, TL'_1, 4);
    Sync:BarrierSync
    \\S1
    DAccess:Load(ShMem, (2 × H_1 + 1) × TL'_2, 4);
    Op:SP((2 × H_1 + 1) × TL'_2);
    DAccess:Store(ShMem, TL'_2, 4);
    Sync:BarrierSync
    \\S2
    DAccess:Load(ShMem, (2 × H_2 + 1) × TL, 4);
    Op:SP((2 × H_1 + 1) × TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

Tile Size and Thread Block Configuration Problem

Tile size and thread mapping problem is depicted in Figure 5.11 and can be reduced to theses questions :

- How to divide the data grid into tiles ?
- How to assign thread blocks to tiles ?
- How thread blocks may be distributed across streaming multiprocessors SMs ?

It is important to do a good choice of the thread block configuration and the tile size processed by each block. |hlTo highlight that, we show an example of performance differences of different configurations of the convolution program on three GPU architectures : (i) GTX 590 (Fermi), (ii) GTX 780 (Kepler) and (iii) GTX 750 (Maxwell).
Each configuration is a tuple of parameters (T_y, T_x, N_y, N_x) where the first two parameters

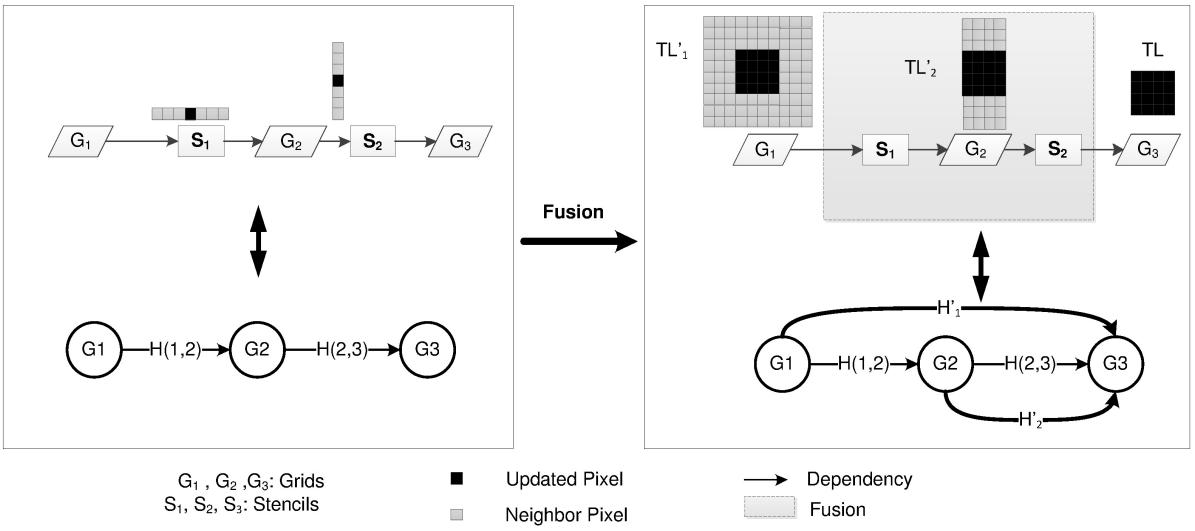


Figure 5.10 Gaussian Blur Fusion

Table 5.2 Gaussian Blur Stencils

Operation	**Stencil**	$\bigcup_i^{\#inGrid} \{inGrid_i : (H_{i,y}, H_{i,x})\}, outGrid$
Convolution on Row	S1	$G1 : (0, r), G2$
Convolution on Col.	S2	$G2 : (r, 0), G3$

define the thread block size, and the two last parameters define the number of elements processed by each thread. The tile size processed by each thread block is given by Equation 5.26. Figure 5.12 shows the execution time of several configurations. We observe that performance vary significantly from a range of configurations to another. Also, we observe that the best configurations set varies from one architecture to another.

$$TL = (T_y \times N_y) \times (T_x \times N_x) \tag{5.26}$$

To deal with the large space of possible configurations and to guide the choice of the best set of configuration, we developed a tool which takes as input :

1. **the halo size** involved for each stencil loaded in the shared memory. These halo sizes are computed with the first stage of the tool described in Section 5.3.2, and ;
2. **the GPU architecture specification** as well as the resource constraints which are summarized in Table 5.3.

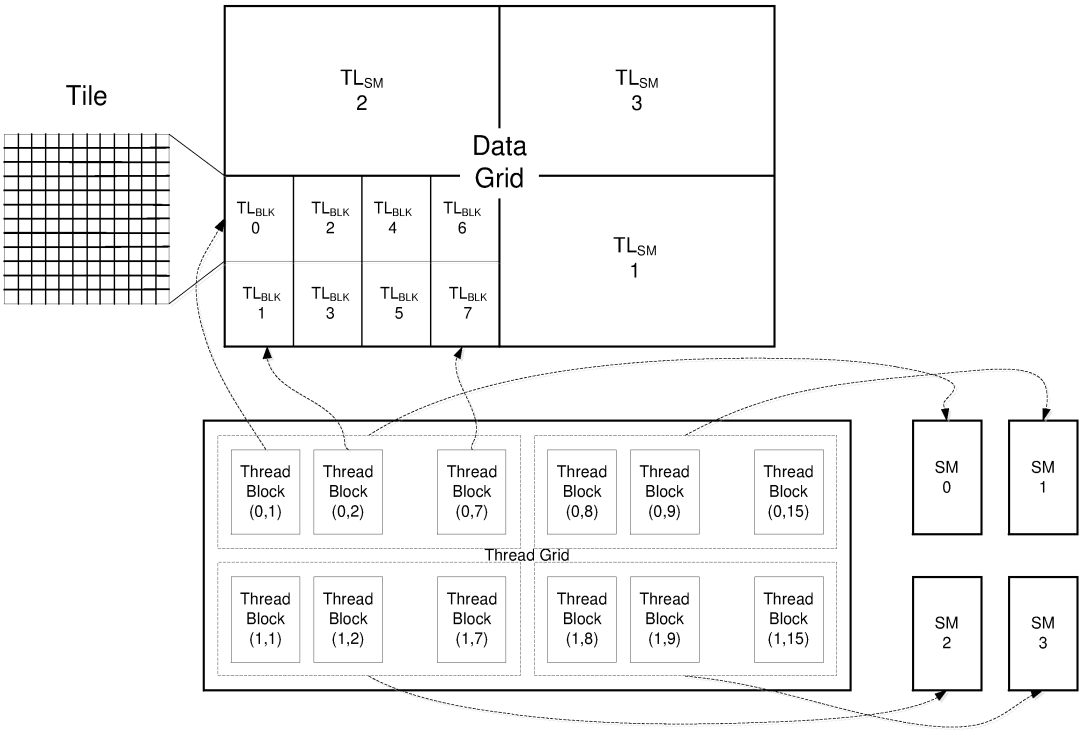


Figure 5.11 Tiling and Thread Block Configuration

This tool offers a coverage of pertinent configurations that may be implemented for a given application on a given GPU architecture. The algorithm responsible to compute the resource usage for each possible configuration is described in Algorithm 2.

For each configuration, the initial tile size is set by Equation 5.26 where, we denote by N_y and N_x respectively, the number of elements processed by each thread at dimension y and x. A thread block is defined by T_y and T_x to denote the $\#Threads$ at each dimension. To augment the initial tile by halo, we use the padding technique which consists on adding extra elements to the halo in order to align the augmented tile size to the thread block size. This technique helps to avoid divergent computation paths and allows better alignment with memory access. Therefore, both T_y and T_x must be greater than the maximum halo size on respectively y and x dimension. Equation 5.27 defines the size of the augmented tile with the needed halo for the stencil computation. The halo added to the initial tile is given by H'_y and H'_x for respectively, y and x dimension. The used shared memory per block is defined in Equation 5.28 where $Smem_{Blk}$ is the total size of the augmented tiles for each input grid. The $\#ResBlk_{SM}$ expressed in Equation 5.29 is determined with respect of the platform constraints and $Smem_{Blk}$.

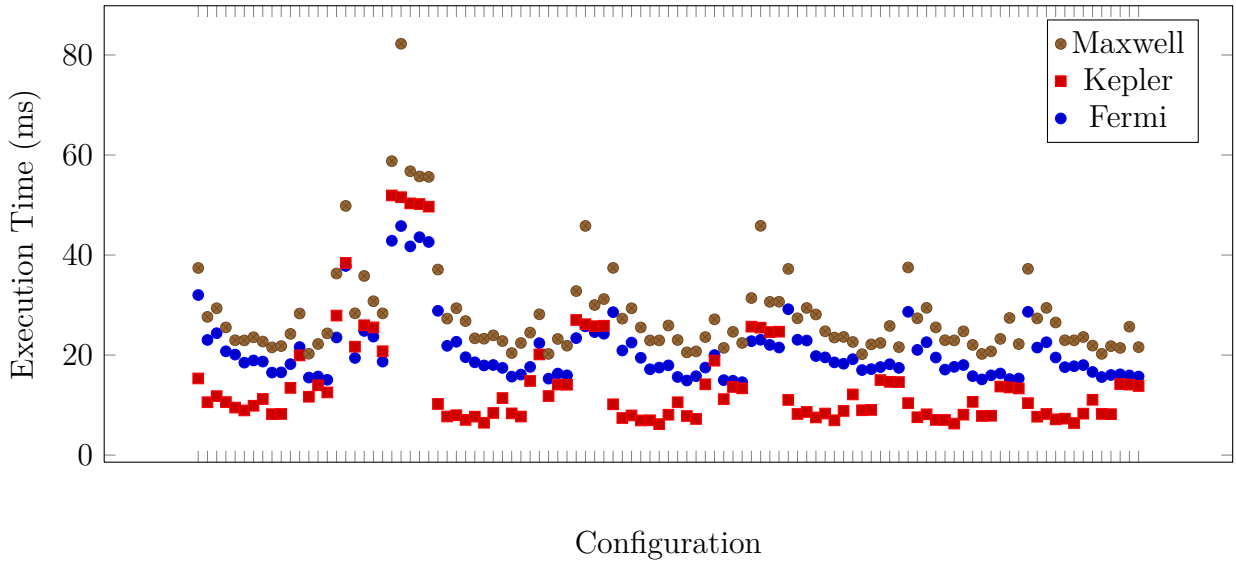


Figure 5.12 Execution Time of Convolution on Rows Kernel (ms)

$$TL'_i = (N_y + \lceil \frac{H'_{i,y}}{T_y} \rceil) \times (N_x + \lceil \frac{H'_{i,x}}{T_x} \rceil) \times (T_y \times T_x) \backslash \forall inGrid_i \tag{5.27}$$

$$Smem_{Blk} = \sum_{i}^{\#inGrids} (TL'_i \times ElemSize(inGrid_i)) \tag{5.28}$$

$$\#ResBlk_{SM} = \min \left(\lfloor \frac{Smem_{SM}^{max}}{Smem_{Blk}} \rfloor, \lfloor \frac{T_{SM}^{max}}{T_{Blk}} \rfloor, \#Blk_{SM}^{max} \right) \tag{5.29}$$

Table 5.3 Platform Specification

Parameter	**Description**	**GTX 590 (2.0)**	**GTX 780 (3.5)**	**GTX 750 (5.0)**
$T_{Blk,dim}^{Max}$	Maximum # of threads per block at the dimension *dim*	1024	1024	1024
T_{Blk}^{Max}	Maximum # of threads per block	1024	1024	1024
T_{SM}^{Max}	Maximum # of threads per SM	1536	2048	2048
Blk_{SM}^{Max}	Maximum # of blocks per SM	8	16	32
$Smem_{Blk}^{Max}$	Maximum ShMem per block	48 KB	48 KB	48 KB
$Smem_{SM}^{Max}$	Maximum ShMem per SM	48 KB	48 KB	64 KB
$\#SM$	Number of SMs	16	12	4

Algorithm 2: Resource Usage Computation Algorithm

Input : $G_i : (H_{i,y}, H_{i,x})$: Halo size needed for the number of tiles in $ShMems$

Input : (IM_y, IM_x) : Data grid size

Input : $\#SM, Smem_{Blk}^{Max}, Smem_{SM}^{Max}, Blk_{SM}^{Max}, T_{Blk}^{Max}, T_{SM}^{Max}$: Architecture specification

foreach *configuration* (T_y, T_x, N_y, N_x) **do**

  if $T_y \geq 2 \times H_y$ ***and*** $T_x \geq 2 \times H_x$ ***and*** $T_y \times T_x \leq T_{Blk}$ **then**

    Compute $Smem_{Blk} = \sum_i TL'_i \times ElemSize(inGrid_i)$;

    Compute $\#ResBlk = Min(\lfloor \frac{Smem_{SM}}{Smem_{Blk}} \rfloor, \#Blk_{SM}^{Max})$;

    Compute $Occupancy = \frac{\#ResBlk \times T_y \times T_x}{T_{SM}^{Max}}$;

    Compute $\#iter_{SM} = \lfloor \frac{IM}{\#SM \times \#ResBlk_{SM} \times TL} \rfloor$;

    Compute $\#RemBlk_{SM} = \lceil \frac{IM - \#iter \times \#ResBlk_{SM} \times \#SM \times TL}{T_{Blk}} \rceil$;

  end

end

5.3.3 Influential Factors on Performance

This section defines the third of our methodology, where we provide a list of main influential factors on GPU performance. These factors are used to setup the performance model. This step is performed only if there is a new used architecture where the impact of the influential parameters on performance vary. In the following the list of the main influential factors on performance :

- **halo size.** each data grid is divided into tiles where each tile is augmented with extra neighbor elements (halo) to respect dependencies at borders. The size of halo depends on the way the grid is split and the properties of computations as depicted previously in Figure 5.3. If we divide the grid according to the orthogonal direction to where the larger number of neighbors is involved, the size of halo increases and yields additional data load and re-computation overheads ;
- **thread occupancy.** Thread occupancy is the rate of actual resident threads by the maximum allowed resident threads per SM (see Equation 5.30). Low thread occupancy prevents both memory latency and arithmetic latency hiding ;
- **branch divergence.** When threads from the same warp follow different paths due to conditional statements, their execution is serialized ;

$$Occupancy = \frac{\#ResBlocks}{Blk_{SM}^{Max}} \tag{5.30}$$

- **memory access coalescing and alignment.** Aligned memory accesses occur when the first address of a device memory is an even multiple of the cache granularity used to service the transaction (either 32 bytes for L2 cache or 128 bytes for L1 cache). Coalesced memory accesses occur when all 32 threads in a warp access a contiguous chunk of memory.
  When threads access non coalesced memory locations we see the performance dropping significantly, and ;
- **Shared Memory Bank Access.** if different parallel threads from the same warp access the same memory bank, the memory access to shared memory is serialized.

5.3.4 Controlled Parameters (Step3)

In this section, we provide the parameters elated to influential factors on performance and that can be controlled by the developer at the programming level. These parameters are :

1. **Data grid decomposition and fusion.** In order to minimize the halo size and increase the effective computation rate, the developer has to manage the data grid decomposition according to the tile size and shape. Another parameter that has an immediate impact on halo size and thread occupancy is the total size of fused grids stored into shared memory. The developer has to select the number and which stencils to fuse in oder to reach high performance computation.
2. **Thread Block Configuration.** HPP runtime for both CUDA and OpenCL organize threads into grids of thread blocks and these blocks may have 1D, 2D or 3D layout. The developer can choose the thread block size and its shape. However this choice depends on several constraints imposed by both hardware architecture and the runtime constraints as mentioned in Section 5.3.2.
   – **Thread block size** - it affects the occupancy rate and by consequence : (i) the degree of concurrency and (ii) both arithmetic operations and memory latency hiding. Thread block size has also an impact on the amount of the remaining work load.
   By selecting small block size, many blocks could be mapped into one SM. This allows better fit with shared memory size when the problem size is not regular and does not match exactly the resource limits. Moreover and since the number of blocks is not distributed equally to the available SMs, small block size gives better load balancing. However, small block size introduces more overhead due to the number of halos which have to be loaded as the number of halos is proportional to the number of block sizes. On the other side, by selecting a large block size, fewer blocks could be mapped into the SM in order to respect the maximum number of threads allowed

per SM. In many cases, this will introduce a bad fit with the resource constraints and by consequence an inefficient use of available computation capability of the hardware platform. However, since the number of loaded halos is proportional to the number of thread blocks, the overhead due to the halo loading and re-computation is minimized.

– **Thread block layout** - it affects the memory access efficiency. The number of threads T_x has an immediate impact on the memory alignment and banked shared memory access efficiency. Examples of memory efficiency impact of T_x for 1D horizontal stencil, 1D vertical stencil and 2D stencil are shown respectively in, Table 5.4 Table 5.5 and Table 5.6. The number of transactions per memory request to both shared memory and global memory is immediately related to T_x. We observe that the number of transaction per request to shared memory doubles when T_x is not multiple of a memory banks number size in 1D horizontal stencil computation. The number of transactions per request to global memory doubles when T_x is not multiple of a half warp and this number increases as T_x decreases. The thread block layout has also an impact on the halo loading and re-computation overhead. Depending on the halo layout imposed by the application, the user has to choose the better block layout that minimizes this overhead and utilizes better the hardware computation capability.

Table 5.4 # Transactions per Request (1D Horizontal Stencil)

T_x	**Shared Load**	**Shared Store**	**Global Load**	**Global Store**
8	2	2	5.72	8
16	2	2	4.83	4
32	1	1	4.35	4.42
64	1	1	4	4
128	1	1	4.29	4

Table 5.5 # Transactions per Request (1D Vertical Stencil)

T_x	**Shared Load**	**Shared Store**	**Global Load**	**Global Store**
4	1	1	7.7	8
8	1	1	4	8
16	1	1	4	4
32	1	1	4	4
64	1	1	4	4

Table 5.6 # Transactions per Request (2D Stencil)

T_x	**Shared Load**	**Shared Store**	**Global Load**	**Global Store**
8	1	1	4.79	8
16	1	1	4.34	4
32	1	1	4.16	4
64	1	1	4.51	4

5.3.5 Performance Model for Stencil Computation (Step 4)

This section defines the fourth step of our methodology (step 4). At this level, we present our performance model used to compute the time cost of stencil based processing on NVIDIA GPUs. This performance model takes two inputs :

1. the program characteristics defined at step 1, the resource usage and the computation amount.
2. the influential parameters on performance defined at step 3 used to set up our performance model

We provide a general performance model that considers primarily the computation, the synchronization and memory operations as basic processing operations like mentioned in our formulation.

The proposed performance model focuses on device processing times.
The processing time (T_{proc}) is defined as a sum of : (i) data access time ($T_{DAccess}$), (ii) synchronization time (T_{Sync}) and operation time (T_{Op}) (see Equation 5.31). The operation time is expressed in Equation 5.32 as a function of operation latency denoted by Lat_{Op}, number of performed operations denoted by $\#Op$ and the operation throughput denoted by $OpTh$. Table 5.8 gives the throughputs in number of operations per clock cycle of main operations for different NVIDIA GPU architectures. Lat_{Op} is equal by default to 22 cycles and 11 cycles for respectively, for GPUs of compute capability 2.0 like Fermi and for GPUs of compute capability higher than 3.x like Kepler and Maxwell.
The operation latency could be expressed as a function of the number of resident warps $\#ResWarp$ per SM (see Equation 5.34). Lat_{Op} is equal to zero if $\#ResWarp$ is greater than a minimum number of resident warps denoted by $\#ResWarp_{min}$. Based on the NVIDIA programming guide, $ResWarp$ has to be at least 24 warps for Fermi and 44 warps for Kepler to hide latency operations. Data access time denoted by $T_{DAccess}$ is expressed in Equation 5.33 as a sum of latency time denoted by $Lat_{DaccessType}$ and the time needed to access $\#Elem \times ElemSize$ Bytes.

$$T_{Proc} = T_{DAccess} + T_{Sync} + T_{Op} \tag{5.31}$$

$$T_{Op} = Lat_{Op} + \frac{\#Op}{OpTh(OpType)} \tag{5.32}$$

$$T^{MemType}_{DAccessType} = Lat^{MemType}_{DAccessType} + \frac{\#Trans^{MemType}_{DAccessType} \times \#Elem \times ElemSize}{BW_{MemType} \times Warp} \tag{5.33}$$

$$\begin{cases} Lat_{Op} = f(\#ResWarp) \\ Lat_{DAccessType} = f(\#ResWarp) \\ \#Trans_{DAccessType} = f(T_x) \end{cases} \tag{5.34}$$

The total processing time on device is given by the processing time of the most loaded SM. First, we express the maximum work load that could be assigned to one SM by using the model given below. The number of thread block assigned to an SM is denoted by $\#Blk_{SM}$ and is expressed in Equation 5.35 as a number of compute iterations denoted by $\#iter_{SM}$ multiplied by the number of resident blocks per SM denoted by ($\#ResBlk_{SM}$) and the number of remaining blocks denoted by $\#RemBlk_{SM}$. The $\#iter_{SM}$ is expressed in Equation 5.36 as the number of iterations needed per SM to compute its portion of the total data grid of size IM at the rate of $\#ResBlk_{SM}$ per computation step where each resident thread block computes one tile of size TL.

$$\#Blk_{SM} = \#iter_{SM} \times \#ResBlk_{SM} + \#RemBlk_{SM} \tag{5.35}$$

$$\#iter_{SM} = \lfloor \frac{IM}{\#SM \times \#ResBlk_{SM} \times TL} \rfloor \tag{5.36}$$

$$\#RemBlk_{SM} = \lceil \frac{IM - \#iter \times \#ResBlk_{SM} \times \#SM \times TL}{T_{Blk}} \rceil \tag{5.37}$$

Table 5.7 Throughput of Main Operations per Clock Cycle (Nvidia, 2012)

Compute Capability	**2.0**	**3.5**	**5.0**
32-bit floating-point add, multiply, multiply-add	32	192	128
64-bit floating-point add, multiply, multiply-add	16	64	1
32-bit square root, special functions	4	32	32
32-bit integer add, subtract	32	160	128
32-bit integer multiply, multiply-add	16	32	32

Table 5.8 Average Memory Latency in Clock Cycles (Nvidia, 2012)

Compute Capability	**2.0**	**3.5**	**5.0**
Global Memory Latency (Clock Cycle)	600	300	-
Shared Memory Latency (Clock Cycle)	16	64	1
Global Memory Clock Rate (MHz)	4008	6008	5000
Global Memory Bus Width (bit)	384	384	128
Theoretical Global Mem. Bandwidth (GB/s)	164	288	80
Shared Mem. Bandwidth (Bytes/clock cycle)	4	16	16

5.4 Experimental Results

In order to validate our framework, we implement two real image processing applications on three NVIDIA GPU architectures using the developed tuning tool. First, we provide the experimental results of a relatively simple application composed of two separable convolution stencils and as example Gaussian Blur application and we compare the results with a an SDK separable convolution implementation. Second, we provide the experimental results of a more complex application involving multiple stencils : Canny Edge Detection.

5.4.1 Experimental Environment

In this section, we present the hardware and the programming models that are used to run our experiments. We use two main hardware platforms as host :

1. 2x AMD Opteron 6128 CPU with total of 16 cores running at 2 GHz, and ;
2. AMD 5800K integrating 4 cores running at 4.3 GHz.

In the first platform, our experiments are performed on Maxwell NVIDIA GTX 750 GPU. In the second platform, our experiments are performed on two NVIDIA architectures : Fermi with NVIDIA GTX 590 and Kepler with NVIDIA GTX 780. As programming model, we use CUDA to program the NVIDIA GPUs.

5.4.2 Case Study : Gaussian Blur

Gaussian Blur is represented as two successive stencils. All the details about the implemented stencils are given in Table 5.9. First, we compare the performance of the configuration obtained by the tuning tool with an SDK implementation and this for two different NVIDIA GPU architectures. The tuned configuration is denoted as Tuned S1 and Tuned S2 for, respectively the stencils S1 and S2. SDK S1 and SDK S2 denotes the stencils implemented in the CUDA SDK with a default thread block configuration. The experiments are performed

on two image sizes : (1) 4096x4096 and (2) 8192x8192. The target GPUs are GTX 590, GTX 780 and GTX 750.

We show that the tuned configuration outperforms the default SDK implementation in all cases. The gain of the tuned implementation over the default SDK implementation varies depending on the architecture and the image size. The gain is more relevant in the case of large images and in particular for GTX 780 compared to little gain for GTX 750. All the experimental results are shown in Figure 5.13.

Next, Figure 5.14 depicts the experimental results of an implementation of both non fused Gaussian and fused Gaussian Blur on three NVIDIA GPU architectures. We observe that the tuned configuration outperforms a basic implementation having the configuration $(T_y, T_x, N_y, N_x) = (16, 16, 1, 1)$ for both non fused and fused versions.

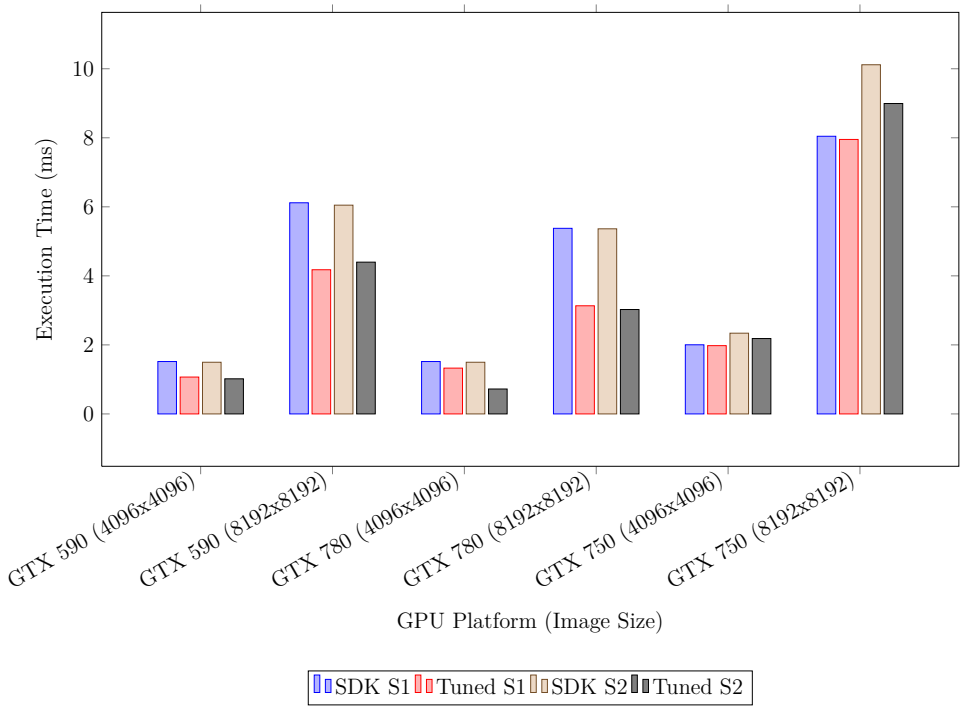


Figure 5.13 Performance Comparison of Different Separate Gaussian Blur Stencils Implementations

Table 5.9 Gaussian Blur Stencils

Operation	Stencil	$\bigcup_i^{\#inGrid} \{inGrid_i : (H_{i,y}, H_{i,x})\}, outGrid$
Convolution on Row	S1	$G1 : (1, 7), G2$
Convolution on Col.	S2	$G2 : (7, 1), G3$
Fused Gauss	(S1,S2)	$G1 : (1, 7), G2 : (7, 1), G3$

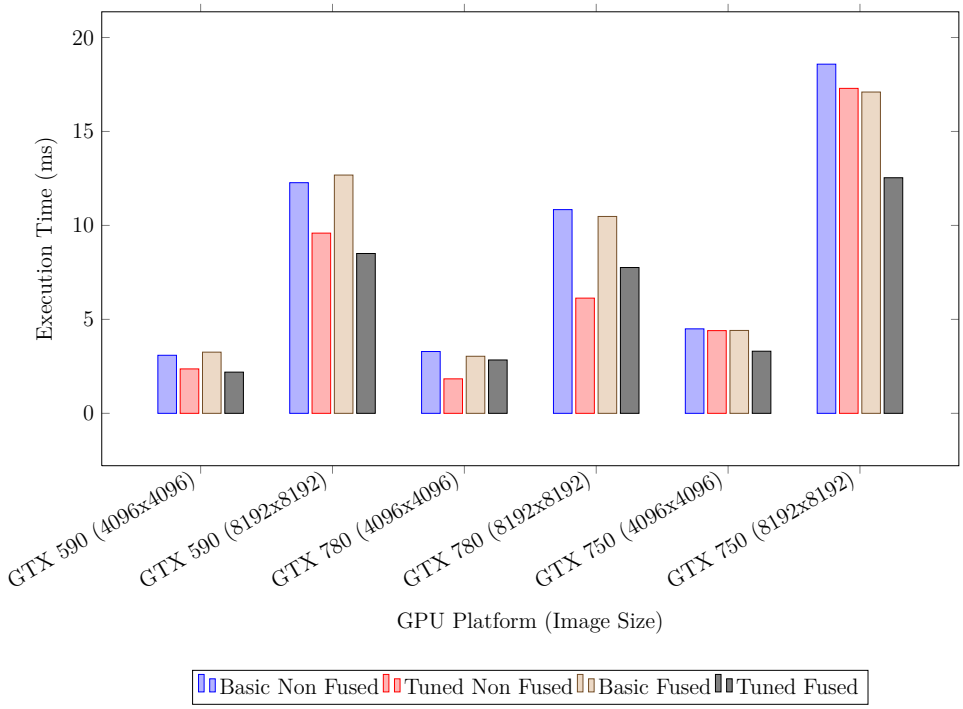


Figure 5.14 Performance Comparison of Different Implementations of Non Fused and Fused Gaussian Blur

5.4.3 Case Study : Canny Edge Detection Application

To evaluate our tuning tool, we study the Canny edge detection (CED) algorithm which can be expressed as a sequence of several stencil operations. Each operation of the CED is expressed as a stencil. A representation of CED under the form of a sequence of stencils is illustrated in Figure 5.15.

The stencil S1 is a convolution on the horizontal direction and the stencil S2 is the convolution in the vertical direction. The sequence of these two stencils form the Gaussian blur stage. The sequence consisting in S3 and S4 and the sequence consisting in S5 and S6 form respectively the gradient at X direction stage and the gradient at Y direction stage. Both S3 and S5 are convolution in the horizontal direction and S4 and S6 are convolution in the vertical direction. The Stencil S7 represents the magnitude calculation stage and the stencil

S8 represents the non maximum suppression. All the stencils parameters are summarized in Table 5.10.

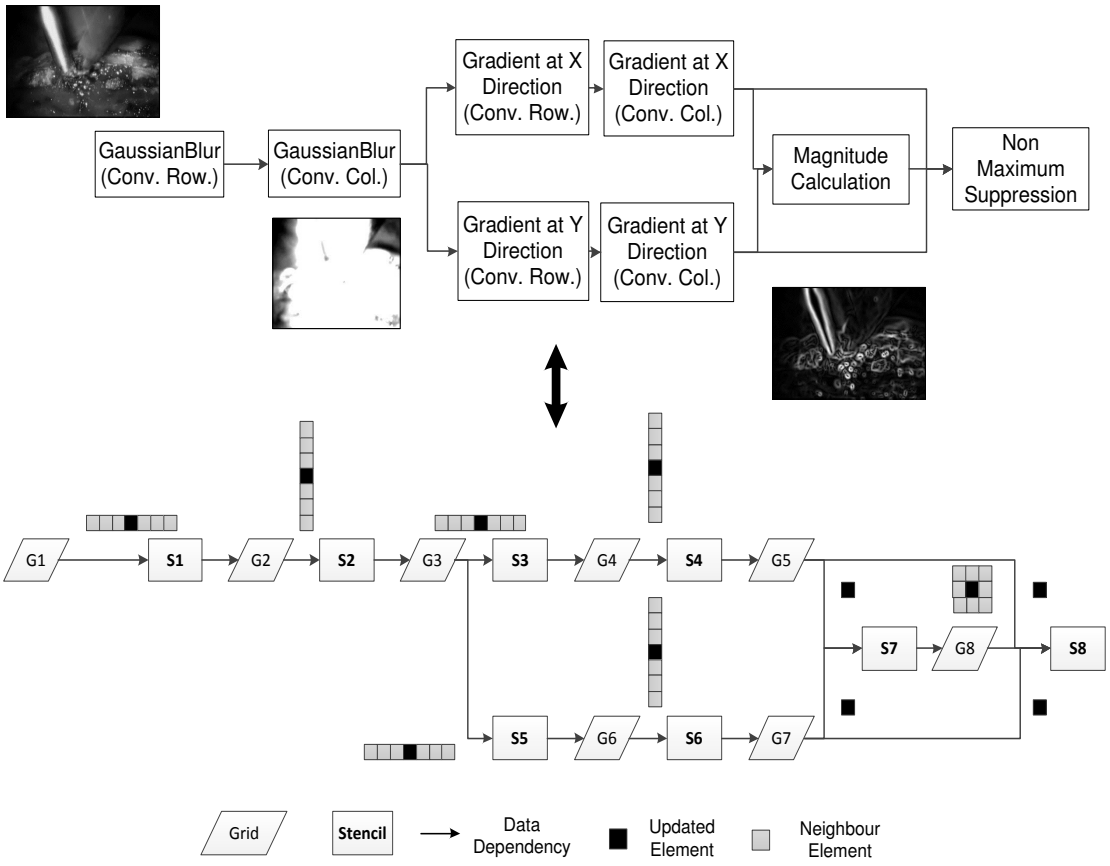


Figure 5.15 Canny Edge Detection Stencils

We evaluate the tuning performance over four fusion combinations of CED with two image sizes (2048x2048) and (4096x4096). The different fusion combinations are summarized in Table 5.11 where stencils between () are the fused stencils and stencils separated by the symbol | are the non fused stencils. we list also the number of grids allocated in the shared memory denoted by $\#shGrids$. For each fusion combination, we compare the performance of a basic implementation and a tuned configuration for three NVIDIA GPU architectures. The results of the performance comparison on GTX 590, GTX 780 and GTX 750 are represented respectively in Figure 5.16-(a), Figure 5.16-(b) and Figure 5.16-(c). As first observation, we show that tuned configuration outperforms the basic implementation in the case of several fusion combinations. We can conclude also that the stencil fusion F2 is the best fusion which

Table 5.10 CED Stencils

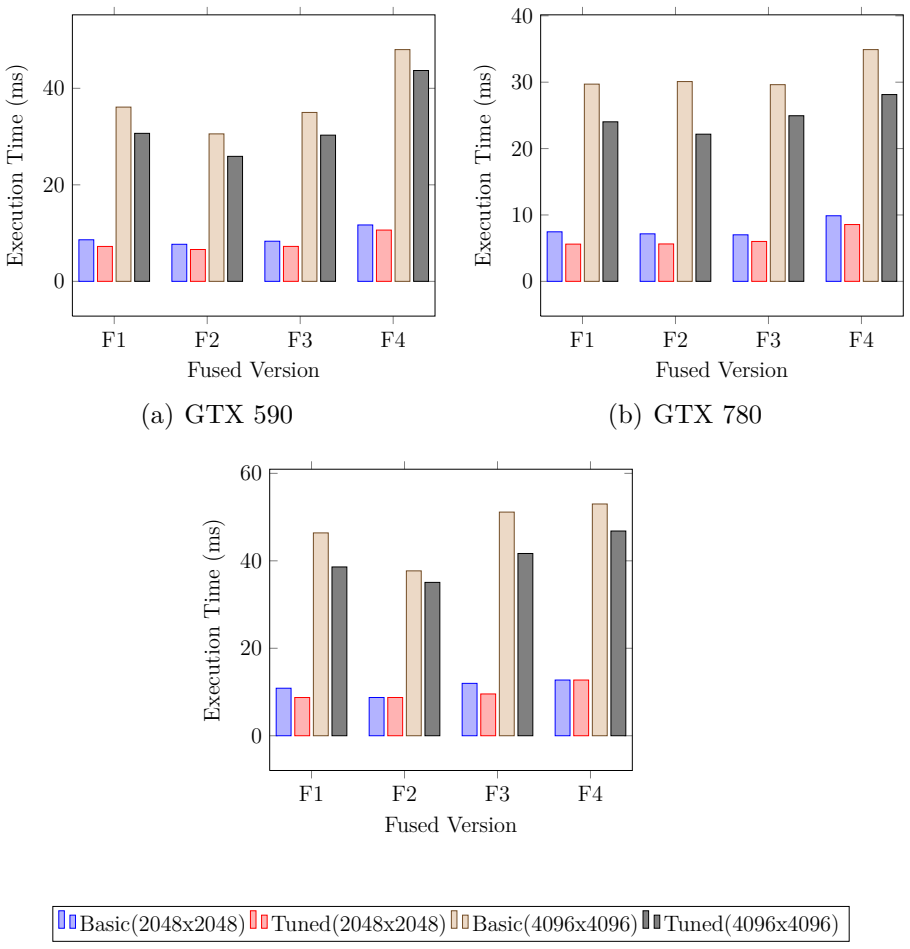
Operation	Stencil	$\bigcup_i^{\#inGrid} \{(inGrid_i, H_i)\}, outGrid$
Gaussian Blur Row	S1	$\{(G1, 0 \times 3)\}, G2$
Gaussian Blur Col.	S2	$\{(G2, 3 \times 0)\}, G3$
Gradient X Row	S3	$\{(G3, 0 \times 3)\}, G4$
Gradient X Col.	S4	$\{(G4, 3 \times 0)\}, G5$
Gradient Y Row	S5	$\{(G3, 0 \times 3)\}, G6$
Gradient Y Col.	S6	$\{(G6, 3 \times 0)\}, G7$
Magnitude	S7	$\{(G5, 0 \times 0), (G7, 0 \times 0)\}, G8$
Non Maximum Suppression	S8	$\{(G5, 0 \times 0), (G7, 0 \times 0), (G8, 1 \times 1)\}, G9$

provides the highest performance for the three GPUs. This is explained by the fact that F2 fuses the largest number of stencils while it allocates the lowest number of grids in the shared memory. The large number of fused stencils improves data locality and avoids the global memory exchanges. Moreover, when the number of allocated grids in the shared memory is small, we can load larger tiles in the shared memory and by consequence reducing the number of computation iterations. This latter is an influent factor on performance.

Based on these observations and the tuning performance results, the developer has the ability to investigate efficiently the performance of several fusion combinations.

Table 5.11 CED Fused Stencils

Fused Version	Fused Stencils	#shGrids
F1	(S1,S2)|(S3,S4)|(S5,S6)|S7|S8	2|2|2|0|0
F2	(S1,S2,S3,S4,S5,S6)|S7|S8	2|0|0
F3	(S1,S2,S3,S4,S5,S6)|(S7,S8)	2|3
F4	(S1,S2,S3,S4,S5,S6,S7,S8)	4



(a) GTX 590

(b) GTX 780

(c) GTX 750

Figure 5.16 Performance Comparison of Basic and Tuned Implementation For Different Fusions on NVIDIA GPUs

5.5 Conclusion

In this chapter, we present our methodology on top of which we build a performance tuning framework for stencil computation. The proposed methodology is based on four major steps :

1. Program formulation,
2. Fusion and tiling configuration,
3. Performance model setup,
4. Projection of program characteristics and configurations on performance model

To automate the process following this methodology, we show a tool that generates all possible implementation configurations and projects each configuration on a given GPU specification to provide the resource usage on that architecture. We believe that this feature is very useful

for the developer to have an early overview of a given program configuration without the need to implement it on the real hardware. The proposed tool supports also the fusion problem where the developer can test the performance of several fusion combinations and the suitable configurations for each without the need to make extra efforts to write a complete code for each combination. In order to find a set of optimal configurations, the tool covers the design space exploration and projects relevant configurations on the proposed performance model. The framework is validated through two concrete image processing applications : (i) the Gaussian Blur filter operation and (ii) the Canny edge detection implemented by performing different fusion combinations and targeting three different NVIDIA GPU architectures.

CHAPTER 6 CONCLUSION AND PERSPECTIVES

This research is motivated by the current context in programming computing intensive applications on modern parallel platforms. Parallel platforms are becoming available to the computational science community with an increasing diversity of architectural models. These platforms increasingly rely on software-controlled parallelism to manage the performance requirements. This results in a daunting problem for performance-oriented programmers. In this last part of this document, we present the summary of the thesis and directions for future research.

6.1 Conclusion

This thesis has addressed three of the main challenges faced currently by the computing intensive application programmers : (1) the evaluation of the different parallel platforms efficiency for a given application (2) the evaluation of the parallel programming strategies and (3) the performance tuning for efficient programming on GPU-based architectures. We focused mainly on computer vision and image processing applications designed on multi-core CPU-based and many-core GPUs-based architectures.

In Chapter 3, we introduced an approach for evaluation of parallel platforms for accelrating computer vision and image processing applications. The evaluation is made for both CPU-based and GPU-based architectures and through three illustrative applications : the Canny Edge Detection, the Shape Refining and Distance Transform. A wide set of platforms was used during the evaluation : a HPC platform based on multicore CPU (Dual Socket), an embedded platform based on multi-core CPU (NVIDIA Tegra K1), three HPC platforms based on manycore GPU (NVIDIA GPU Fermi, NVIDIA GPU Kepler, NVIDIA GPU Maxwell) and two embedded platforms based on manycore (NVIDIA Tegra K1 et STHORM provided by STMicroelectronics).

In Chapter 4, we presented a quantitative analysis of a large variety of parallelization strategies used in programming for parallel platforms. We considered the parallelism granularity, the parallelization models as well as the main programming models for the studied platforms. This analysis is realized on two of the most common parallel platforms : multicore CPU platforms and manycore GPU based platforms, using the Canny Edge Detection application. The

multicore CPU target platform is a dual AMD Opteron 6128 running at 2 GHz. The manycore GPU target platform is a NVIDIA GeForce GPU specifically a Fermi-series graphics processor GTX 480 which integrates 480 SPs distributed on 15 Streaming Multiprocessors (SMs) as 32 Streaming Processors (SPs) per SM. In this chapter, we also outlined a set of guidelines for the efficient parallelization of common applications.

In Chapter 5, we proposed a performance tuning framework enabling programmers to efficiently design applications while leveraging the GPU architectures. This framework is based on a rigorous formulation and a set of algorithms designed to optimize the kernel fusion process. The emphasis was put on HPP platforms that follow host-device architectural model where the host corresponds to a multicore CPU and the device corresponds to a manycore GPU.

6.2 Perspectives

The results presented in this dissertation point out to several interesting directions for future work :

1. The presented research work studied a wide set of platforms. However, we did not explore the AMD GPU-based platform and the Hybrid parallel platforms integrating CPU and GPU in the same die like AMD APU. An interesting research direction is to investigate their efficiency and if there are some programming strategies more suitable for such platforms than discrete GPU platforms.
2. Several research lines are open by the performance tuning framework proposed in Chapter 5 :
   - While we claim that the framework can be applied on 3D grids and complex applications, more experiments are needed to solidify our hypotheses.
   - The analysis of the accuracy for the proposed performance model needs to be addressed.
   - The automation of the fusion process using heuristics would be a further improvement of our framework.
3. Finally, it is our intention to parallelize a full complex application integrating the algorithms analyzed in this work. The target application is the augmented reality system for minimally invasive surgery.

REFERENCES

G. Bradski, “The openCV library”, *Dr Dobbs Journal of Software Tools*, vol. 25, no. 11, pp. 120–126, 2000. En ligne : `http://opencv.willowgarage.com`

A. Z. Brethorst, N. Desai, D. P. Enright, et R. Scrofano, “Performance evaluation of canny edge detection on a tiled multicore architecture”, dans *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, série Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 7872, jan 2011.

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, et P. Hanrahan, “Brook for gpus : stream computing on graphics hardware”, dans *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3. ACM, 2004, pp. 777–786.

J. M. Bull et D. O’Neill, “A microbenchmark suite for openMP 2.0”, *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 41–48, Déc. 2001.

D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997.

H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, et I. Said, “Evaluation of successive CPUs/APUs/GPUs based on an openCL finite difference stencil”, dans *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 405–409.

J. Canny, “A computational approach to edge detection”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, et K. Skadron, “A performance study of general-purpose applications on graphics processors using cuda”, *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008. DOI : `http://dx.doi.org/10.1016/j.jpdc.2008.05.014`

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, et K. Skadron, “Rodinia : A benchmark suite for heterogeneous computing”, dans *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.* IEEE, Oct. 2009, pp. 44–54. DOI : `10.1109/IISWC.2009.5306797`. En ligne : `http://dx.doi.org/10.1109/IISWC.2009.5306797`

T. P. Chen, D. Budnikov, C. J. Hughes, et Y.-K. Chen, “Computer vision on multi-core processors : Articulated body tracking”, *IEEE International Conference on Multimedia and Expo*, pp. 1862–1865, 2007.

T. P.-c. Chen, D. Budnikov, C. J. Hughes, et Y.-K. Chen, “Computer vision on multi-core processors : Articulated body tracking”, dans *Multimedia and Expo, 2007 IEEE International Conference on*. IEEE, 2007, pp. 1862–1865.

J. Cheng, M. Grossman, et T. McKercher, *Professional CUDA C Programming*. John Wiley & Sons, 2014.

S. Cook, *CUDA programming : a developer's guide to parallel computing with GPUs*. Newnes, 2012.

L. Dagum et R. Menon, “OpenMP : an industry standard api for shared-memory programming”, *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, et K. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures”, dans *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 4.

V. V. Dimakopoulos, P. E. Hadjidoukas, et G. C. Philos, “A microbenchmark study of openMP overheads under nested parallelism”, dans *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, série IWOMP'08. Berlin, Heidelberg : Springer-Verlag, 2008, pp. 1–12. En ligne : `http://dl.acm.org/citation.cfm?id=1789826.1789828`

A. Djabelkhir et A. Seznec, “Characterization of embedded applications for decoupled processor architecture”, dans *Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop on*. IEEE, 2003, pp. 119–127.

P. Eberhart, I. Said, P. Fortin, et H. Calandra, “Hybrid strategy for stencil computations on the apu”, dans *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*, 2014, pp. 43–49.

A. Eklund, P. Dufort, D. Forsberg, et S. M. LaConte, “Medical image processing on the gpu–past, present and future”, *Medical image analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

A. Ensor et S. Hall, “GPU-based image analysis on mobile devices”, *CoRR*, 2011.

R. Fabbri, L. D. F. Costa, J. C. Torelli, et O. M. Bruno, “2d euclidean distance transform algorithms : A comparative survey”, *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 2, 2008.

M. Frigo, C. E. Leiserson, et K. H. Randall, “The implementation of the Cilk-5 multithreaded language”, *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, may 1998. DOI : 10.1145/277652.277725. En ligne : http://doi.acm.org/10.1145/277652.277725

M. I. Gordon, W. Thies, et S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs”, dans *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 151–162.

R. L. Graham, T. S. Woodall, et J. M. Squyres, “Open MPI : A flexible high performance MPI”, dans *Parallel Processing and Applied Mathematics*. Springer, 2006, pp. 228–239.

T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, et S. Verdoolaege, “Split tiling for gpus : automatic parallelization using trapezoidal tiles”, dans *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 24–31.

O. W. Group *et al.*, “The openacc application programming interface”, 2011.

J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzarán, et D. A. Padua, “Programming with tiles”, dans *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, S. Chatterjee et M. L. Scott, éds. ACM, 2008, pp. 111–122. DOI : http://doi.acm.org/10.1145/1345206.1345225

A. Jerraya et W. Wolf, *Multiprocessor systems-on-chips*. Elsevier, 2004.

S. A. Kamil, “Productive high performance parallel programming with auto-tuned domain-specific embedded languages”, DTIC Document, Rapp. tech., 2013.

K. Karimi, N. G. Dickson, et F. Hamze, “A performance comparison of CUDA and openCL”, *CoRR*, vol. abs/1005.2581, 2010.

P. Kegel, M. Schellmann, et S. Gorlatch, “Comparing programming models for medical imaging on multi-core systems”, *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 10, pp. 1051–1065, Juil. 2011.

——, “Comparing programming models for medical imaging on multi-core systems”, *Concurrency and Computation : Practice and Experience*, vol. 23, no. 10, pp. 1051–1065, 2011.

S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, et P. Sadayappan, “Effective automatic parallelization of stencil computations”, dans *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 235–244.

V. Kumar, A. Grama, A. Gupta, et G. Karypis, *Introduction to parallel computing : design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

S. Lee, S.-J. Min, et R. Eigenmann, “OpenMP to GPGPU : a compiler framework for automatic translation and optimization”, *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, et J. D. Owens, “Glift : Generic, efficient, random-access GPU data structures”, *ACM Transactions on Graphics (TOG)*, vol. 25, no. 1, pp. 60–99, 2006.

C. E. Leiserson, “The Cilk++ concurrency platform”, dans *Proceedings of the 46th Annual Design Automation Conference*, série DAC ’09. New York, NY, USA : ACM, 2009, pp. 522–527. DOI : 10.1145/1629911.1630048. En ligne : http://doi.acm.org/10.1145/1629911.1630048

Z. Li et Y. Song, “Automatic tiling of iterative stencil loops”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 6, pp. 975–1028, 2004.

Y. Luo et R. Duraiswami, “Canny edge detection on NVIDIA CUDA”, dans *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW’08. IEEE Computer Society Conference on*. IEEE, 2008, pp. 1–8.

T. Lutz, C. Fensch, et M. Cole, “Partans : An autotuning framework for stencil computation on multi-gpu systems”, *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 59, 2013.

W. R. Mark, R. S. Glanville, K. Akeley, et M. J. Kilgard, “Cg : a system for programming graphics hardware in a c-like language”, dans *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 896–907.

A. Marowka, “Empirical analysis of parallelism overheads on CMPs”, dans *Proceedings of the 8th international conference on Parallel processing and applied mathematics : Part I*, série PPAM’09, 2010, pp. 596–605.

MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts : The MathWorks Inc., 2010.

P. S. McCormick, J. Inman, J. Ahrens, C. Hansen, et G. Roth, “Scout : A hardware-accelerated system for quantitatively driven visualization and analysis”, dans *Visualization, 2004. IEEE*. IEEE, 2004, pp. 171–178.

D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, et D. Dutoit, “Platform 2012, a many-core computing accelerator for embedded socs : performance evaluation of visual analytics applications”, dans *Proceedings of the 49th Annual Design Automation Conference*, série DAC ’12. New York, NY, USA : ACM, 2012, pp. 1137–1142.

J. Meng et K. Skadron, “A performance study for iterative stencil loops on GPUS with ghost zone optimizations”, *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011. DOI : `http://dx.doi.org/10.1007/s10766-010-0142-5`

——, “Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs”, dans *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 256–265.

A. Munshi, D. Ginsburg, et D. Shreiner, *OpenGL(R) ES 2.0 Programming Guide*, 1er éd. Addison-Wesley Professional, 2008.

C. A. Navarro, N. Hitschfeld-Kahler, et L. Mateu, “A survey on parallel computing and its applications in data-parallel problems using gpu architectures”, *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

T. Ni, “Direct compute : Bring gpu computing to the mainstream”, dans *GPU Technology Conference*, 2009.

G. Nicolescu et P. J. Mosterman, *Model-based design for embedded systems*. CRC Press, 2009.

S. Niu, J. Yang, S. Wang, et G. Chen, “Improvement and parallel implementation of canny edge detection algorithm based on gpu”, dans *ASIC (ASICON), 2011 IEEE 9th International Conference on*, 2011, pp. 641–644.

NVIDIA, “Whitepaper nvidia’s next generation CUDA compute architecture : Fermi”, Rapp. tech., 2009.

——, “Whitepaper nvidias next generation cuda compute architecture : Kepler GK110”, Rapp. tech., 2012.

——, “Whitepaper nvidia geforce gtx ti featuring first-generation maxwell gpu technology, designed for extreme performance per watt”, Rapp. tech., 2014.

——, “Whitepaper NVIDIA Tegra a new era in mobile computing”, Rapp. tech., 2013.

C. Nvidia, “Compute unified device architecture programming guide”, 2007.

——, “C programming guide v5. 0”, *NVIDIA Corporation, October*, 2012.

K. Ogawa, Y. Ito, et K. Nakano, “Efficient canny edge detection using a gpu”, dans *Proceedings of the 2010 First International Conference on Networking and Computing*, série ICNC ’10. Washington, DC, USA : IEEE Computer Society, 2010, pp. 279–280.

S. Olivier et J. Prins, “Comparison of openMP 3.0 and other task parallel frameworks on unbalanced task graphs”, *International Journal of Parallel Programming*, vol. 38, pp. 341–360, 2010, 10.1007/s10766-010-0140-7. En ligne : `http://dx.doi.org/10.1007/s10766-010-0140-7`

S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, et C.-W. Tseng, “Uts : an unbalanced tree search benchmark”, dans *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, série LCPC’06. Berlin, Heidelberg : Springer-Verlag, 2007, pp. 235–250. En ligne : `http://dl.acm.org/citation.cfm?id=1757112.1757137`

S. Orlando, P. Palmerini, et R. Perego, “Mixed data and task parallelism with HPF and PVM”, *Cluster Computing*, vol. 3, no. 3, pp. 201–213, 2000.

S. Ourselin, R. Stefanescu, et X. Pennec, “Robust registration of multi-modal images : towards real-time clinical applications”, dans *Medical Image Computing and Computer-Assisted Intervention—MICCAI 2002*. Springer, 2002, pp. 140–147.

P. S. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann, 1997.

V. K. Pallipuram, M. Bhuiyan, et M. C. Smith, “A comparative study of GPU programming models and architectures using neural networks”, *The Journal of Supercomputing*, vol. 61,

no. 3, pp. 673–718, 2012.

T. Patil, “Evaluation of multi-core architectures for image processing algorithms”, 2009.

P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné *et al.*, “Parallel programming models for a multiprocessor soc platform applied to networking and multimedia”, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 7, pp. 667–680, 2006.

C. Peeper et J. L. Mitchell, “Introduction to the directx® 9 high level shading language”, *ShaderX2 : Introduction and Tutorials with DirectX*, vol. 9, 2003.

C. Pheatt, “Intel® threading building blocks”, *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

J. A. Pienaar, A. Raghunathan, et S. Chakradhar, “Mdr : performance model driven runtime for heterogeneous parallel platforms”, dans *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 225–234.

F. Pratas, P. Trancoso, A. Stamatakis, et L. Sousa, “Fine-grain parallelism using multi-core, Cell/BE, and GPU systems : Accelerating the phylogenetic likelihood function”, dans *Proceedings of the 2009 International Conference on Parallel Processing*, série ICPP ’09. Washington, DC, USA : IEEE Computer Society, 2009, pp. 9–17. DOI : 10.1109/ICPP.2009.30. En ligne : http://dx.doi.org/10.1109/ICPP.2009.30

J. Protic, M. Tomasevic, et V. Milutinović, *Distributed shared memory : Concepts and systems*. John Wiley & Sons, 1998, vol. 21.

J. Rodríguez-Rosa, A. J. Dorta, C. Rodríguez, et F. de Sande, “Exploiting task and data parallelism”, dans *Proc. of the 5th European Workshop on OpenMP (EWOMP 2003)*, Aachen, Germany, Sep 2003, pp. 107–116.

A. Rosenfeld et J. L. Pfaltz, “Sequential operations in digital picture processing”, *Journal of the ACM (JACM)*, vol. 13, no. 4, pp. 471–494, 1966.

R. J. Rost, B. M. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, et M. Weiblen, *OpenGL shading language*. Pearson Education, 2009.

M. Sonka, V. Hlavac, et R. Boyle, *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

J. E. Stone, D. Gohara, et G. Shi, “OpenCL : A parallel programming standard for heterogeneous computing systems”, *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

K. Swargha et P. Rodrigues, “Performance amelioration of edge detection algorithms using concurrent programming”, *International Conference on Modeling Optimization and Computing*, vol. 38, pp. 2824 – 2831, 2012.

S. Tabik, A. Murarasu, et L. F. Romero, “Evaluating the fission/fusion transformation of an iterative multiple 3d-stencil on gpus”, *HiStencils 2014*, p. 81, 2014.

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, et C. E. Leiserson, “The pochoir stencil compiler”, dans *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 117–128.

M. Team, “Mvapich2 1.9 user guide”, 2001.

W. Thies, M. Karczmarek, et S. Amarasinghe, “Streamit : A language for streaming applications”, dans *Compiler Construction*. Springer, 2002, pp. 179–196.

X. Tian, J. P. Hoeflinger, G. Haab, Y.-K. Chen, M. Girkar, et S. Shah, “A compiler for exploiting nested parallelism in openMP programs”, *Parallel Comput.*, vol. 31, no. 10-12, pp. 960–983, oct 2005.

S. Williams, A. Waterman, et D. Patterson, “Roofline : an insightful visual performance model for multicore architectures”, *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

L. Windisch, F. Cheriet, et G. Grimard, *Bayesian differentiation of multi-scale line-structures for model-free instrument segmentation in thoracoscopic images*. Springer, 2005.

H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, et S. Chakradhar, “Optimizing data warehousing applications for gpus using kernel fusion/fission”, dans *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 2433–2442.